MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: SOFTWARE ENGINEERING

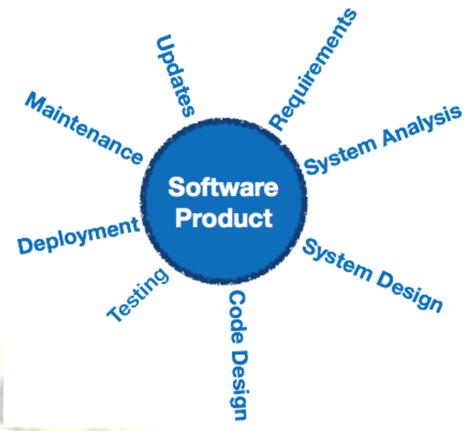
SUBJECT CODE: SEE6G

SEMESTER: VI

PREPARED BY: PROF. A. JEROMEROBINSON

E-CONTENT FOR SOFTWARE ENGINEERING

Unit 1:
Introduction to Software Engineering



Let us understand what Software Engineering stands for. The term is made of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Some definitions

IEEE defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in the above statement.

Fritz Bauer, a German computer scientist, defines software engineering as:

"Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines."

We have defined software engineering as the technological discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates. Software engineering differs from traditional computer programming in that engineering - like techniques are used to specify, design, implement, validate and maintain software products within the time and budget constraints established for the project. In addition software engineering is concerned with managerial issues that lie outside the domain of traditional programming on small projects, perhaps involving one or two programmers for one or two months, the issues of concern are primarily technical in nature. On projects involving more programmers and longer time durations, management control is required to coordinate the technical activities.

In this text the term "Programmer" is used to denote an individual who is concerned with the details of implementing, packaging and modifying algorithms and data structures written in particular programming languages. Software engineers are additionally concerned with issues of analysis, design, verification and testing, documentation, software maintenance and project management. A software engineer should have considerable skill and experience as a programmer in order to understand the problem areas, goals and objectives of software engineering.

It is sometimes said that software engineering concepts are applicable only to large project of long duration. On large projects standard practices and formal procedures are essential and some of the notations, tools and techniques of software engineering have been developed specifically for large projects. The fundamental concepts of software development and maintenance presented in this text are useful on every programming project; however, a few of the techniques discussed are cost-effective only on large projects.

The term "Computer software" is often taken to be synonymous with "Computer program" or "Source code". In this text, "Computer software" is synonymous with "Software product". Thus computer software includes the source code and all the associated documents and documentations that constitute a software product. Requirements, documents design specifications, source code, test plans, principles of operation, quality assurance procedures, user's manuals, installation instructions and training aids are all components of software product. Software products include system -level software as well as application software developed to solve specific problems for end users of computing systems.

In this text "documentation" explains the characteristics of a document. Internal documentation of source code describes the characteristics of the code and external documentation explains the characteristics of the documents associated with the code. Software engineering is concerned with systematic development and maintenance of documentation and supporting documents as well as the source code.

In this text the terms "developer" and "software engineer" are used inter changeably. The term "customer" is used to denote an individual or organisation that initiates procurement or modification of a software product.

Software quality is a primary concern of software engineers. Quality attributes of importance for any particular software product are of course dependent on the nature of that product. In some instances transportability of the software product between machines may be an attribute of prime importance while efficient utilization of memory space may be paramount in other cases. There are however a few fundamental quality attributes that every software product should possess. These include usefulness, clarity, reliability, efficiency and cost-effectiveness.

The most important quality attribute a software product can possess is usefulness, i.e., the software product must satisfy user needs. This may seem painfully obvious but delivered software products frequently do not perform the functions expected of them. The problem is symptomatic of poor communication among the customer, the users and the software engineers, careful planning, analysis and customer involvement are required to develop careful software products.

Software reliability is "the ability of a program to perform a required function under stated conditions for a stated period of time". The degree of reliability required of a particular product can be expressed in terms of the cost of product failure. The amount to be spent on attaining increased reliability is a function of the cost of product failure: however there is a minimal level of reliability that every software product must possess.

Software products must be clearly written and easy to understand. Testing and maintenance activities consume a large portions of most software budgets. The key to making a system testable and maintainable is to make it understandable. Software products must also exhibit conceptual integrity and clarity of purpose to the users. Many of the techniques discussed in this text have the goal of improving both the internal and external clarity of software products.

A software product should be efficient - but only as appropriate for the particular application. In the early days of digital computers hardware was very expensive and very slew^by today's standards; efficiency was the primary quality attribute of computer programs. As software becomes larger and more complex, attributes such as usefulness, reliability and clarity assume overriding importance for most software products.

On the other hand, there are software products (ie. real time systems implemented on microprocessors) that are critically constrained in memory space and execution time. Efficiency remains a primary quality attribute for those systems.

Finally, a software product must be cost-effective in development, in maintenance and in use. Development and maintenance efforts devoted to increasing the efficiency and reliability of a software product must be appropriate to the intended application of the product. In use, a software product must perform an existing task using less time or human and machine resources that were required without the product or it must provide new services that were not feasible without it. Sometimes, a new software product will not perform the intended tasks as expected, but will provide unforseen options and capabilities that make it attractive on different grounds than originally envisioned.

Some size factors

Estimation of the size of software is an essential part of Software Project Management. It helps the project manager to further predict the effort and time which will be needed to build the project. Various measures are used in project size estimation. Some of these are:

- Lines of Code
- Number of entities in ER diagram
- Total number of processes in detailed data flow diagram
- Function points

1. Lines of Code (LOC): As the name suggest, LOC count the total number of lines of source code in a project. The units of LOC are:

- KLOC- Thousand lines of code
- NLOC- Non comment lines of code
- KDSI- Thousands of delivered source instruction

The size is estimated by comparing it with the existing systems of same kind. The experts use it to predict the required size of various components of software and then add them to get the total size.

Advantages:

- Universally accepted and is used in many models like COCOMO.
- Estimation is closer to developer’s perspective.
- Simple to use.

Disadvantages:

- Different programming languages contains different number of lines.
- No proper industry standard exist for this technique.
- It is difficult to estimate the size using this technique in early stages of project.

In this section, the level of effort devoted to software development and maintenance is discussed; the distribution of effort among activities is presented; and the size categories for software projects are described. The results reported here summarize many different software projects from many different organisations, and they should be viewed only in the statistical sense.

1.2.1. Total effort Devoted to software

Current demand for software technologists exceeds the available supply and it is estimated that demand will exceed supply by 750,000 to 2,000,000 people in the future. Thus, a major goal of software engineering is to provide tools and techniques to increase the productivity of the available software engineers.

Figure 1.1 illustrates the changing ratio of expenditures for hardware and software over time. In 1960, the ratio was approximately 80 percent hardware cost to 20 percent software cost. By 1980, the ratio was reversed : approximates 20 percent hardware cost to 80 percent software cost. Also observe that software maintenance is a large and growing portion of the software effort.

Reasons for the trends illustrated in Fig. 1.1. are not hard to determine. Transistors, integrated circuits and VLSI have resulted in dramatic decreases in hardware costs. On the other hand software is labour-intensive and personal costs are constantly increasing. Similarly software maintenance is an increasing portion of software cost because with passing time more software accumulates.

The figure extrapolates overall hardware / software cost trends in the united states from an Air Force Study conducted in 1972. It should not be interpreted as the relative cost of hardware and software for any particular'computing system or development project. Another qualification to figure 1.1.concerns the projected level of software maintenance costs. Use of modern software maintenance costs. Use of modern software engineering tools and techniques should result n software products that are easier to maintain :
however, the increasing number, size and complexity of software products may offset the gains from modern technology.

1.2.2. Distribution of Effort

The typical lifespan for a software product is 1 to 3 years in development and 5 to 15 years in use-(maintenance). The distribution of effort between development and maintenance has been variously reported as 40/60, 30/70 and even 10/90. This is not surprising when it is understood that maintenance comprises all activities following initial release of a software product.
Software maintenance involves three types of activities enhancing the capabilities of the product, adapting the product to new processing environments and correcting bags. Typical distributions of maintenance effort for enhancement, adaptation and correction are 60 percent, 20 percent and 20 percent respectively. During the development phase of a software product, the distribution of effort is typically 40 percent for analysis and design; 20 percent-for implementation, debugging, and unit testing; and 40 percent for integration and acceptance testing. Taking the distribution of effort between development and maintenance to be 40/60 and normalizing total effort to 100 percent results in the distributions presented in fig. 1.2.

Quality and productivity factors

Software quality and programmer productivity can be improved by improving the processes used to develop and maintain software products. Some factors that influence quality and productivity are listed in Table 1.2 ~

Table 1.2 Factors that influence quality and productivity

Problem under standing
Individual ability Stability of requirements
Team communication Required skills
Product complexity Facilities and resources
Appropriate notations Adequacy of training
Systematic approaches Managements skills
Change control Appropriate goals
Level of technology Rising expectations Required reliability
Available time

Individual ability

Production and maintenance of software products are labour-intensive activities. Productivity and quality are thus direct functions of individuals ability and effort. There are two aspects to ability the general competence of the individual and familiarity of the individual with the particular application area. Competent data processing programmers are not usually competent in scientific piogrammers ordinarily competent system programmers. Lack of familiarity with, the application area can result in law productivity and poor quality.

On very large and extremely large projects, the number of programmers is so large that individual differences in programmer productivity will tend to average out. However, the modules developed by weaker programmers may exhibit poor quality and may lag in delivery time. Small and medium-size projects are extremely sensitive to the ability of the individualprogrammer.

In programming, the general guideline is to utilize outstanding people. However, it is not always possible to hire exceptional individuals. One of the goals of software engineering is to provide notations tools and techniques that will enable programmers of good but not outstanding ability to perform their work activities in a competent, professional manner. By investing, in better hardware and software tools, organizations can shift software engineering from labour-intensive to a Capital-intensive industry. Nevertheless, individual ability will be a primary factor in quality and productivity into the future.

Team Communication

Programming has traditionally been regarded as an individual and private activity. Many programmers have low social needs and prefer to work alone. Programs are seldom perceived as public documents and programmers seldom discuss the exact details of their work in systematic manner. As a result, it is possible for programmers to misunderstand the role of their modules in an evolving system and to make mistakes that may not be detected until some time later. Many of the recent invocations in software engineering, such as design reviews structural walk throughs and code-reading exercises, have the goals of making software more invisible and improving communication among programmers.

Increasing product size result in decreasing programmer productivity due to the increased complexity of interactions among program components and due to the increased communication required among programmers, managers and customers. Increasing the number of team members, increases the number of communications pathsr-Also, each team member must learn the project and overcome the "learning curve" effect before becoming a contributing team member. This further increases overhead and lowers overall productivity.

Many processes in software development are interdependent and must occur in sequential order. Adding more programmers to an on going project may be counter productive unless there are independent tasks that the new programmers can perform without incurring the overhead required to learn various details of the project. Adding more programmers to a late project may make it later.

Product complexity

There are three generally acknowledged levels of product complexity; applications programs, utility programs and system level programs. Application programs include scientific and data processing routine written in a high-level language., such as FORTRAN. Pascal or COBOL. Utility programs include compilers, assemblers, linkage editors and loaders. They may be written in a high-level language such as Pascal or Ada or in assembly language. Systems programs include data communication packages, real-time process control systems and operating systems routines which are usually written in assembly language or a high-level systems language such as PL/1 or Ada.

Applications programs have the highest productivity and systems programs the lowest productivity, as measured in lines of code per programmer day. Utility programs can be produced at a rate 5 to 10 times and application programs at a rate 5 to 10 times that of systems programs. The effort required to develop and maintain a software product is a non linear function of product size arid complexity.

VARIATIONS IN PROGRAMMER PRODUCTIVITY

Performance Measure	**Ratio**	
Debugging hours # 1	**28**	**1**
Debugging hours #'2	**26**	**1**
CPU Sec. for DVMT # 1	**8**	**1**
CPU Sec. for DVMT # 2	**11**	**1**
Coding hours # 1	**16**	**1**
Coding hours # 2	**25**	**1**
Program Size # 1	**6**	**1**
Program Size # 2	**5**	**1**

Managerial issue.

Managerial activities are as important as the technical activities for the success of a software product. Managers control the resources and the environment in which technical activities occur. Managers also have the responsibility for ensuring that software products are delivered on time and within cost estimates and that products exhibit the functional and quality attributes desired by the customer. Other management responsibilities include developing business plans, recruiting the customers, developing marketing strategies and recruiting and training employees.

Some of the management problems that are to be solved are listed below:

1. Planningfor software engineering projects.
2. Procedures and techniques for the selection of project managers are poor.
3. The accountability of many software engineering projects is poor leaving some question as to who is responsible for various project functions.
4. The ability to accurately estimate the resources required to accomplish a software development project is poor.
5. Success criterion for software development projects are frequently inappropriate. This results in software products that are unreliable difficult to use and difficult to maintain.
6. Decision rules to aid in selecting the proper organizational structure are not available.

1. Decision rules to aid in selecting the correct management techniques for .software engineering projects are not available.

8. Procedures methods and techniques for designing a project control system that will enable project managers to successful control their project are not readily available.
9. Procedures, techniques, strategies and aids that will provide visibility of progress to the project manager are not available.
10. Standards and techniques for measuring the quality of performance and the quality of production expected from programmers and data processing analysts are not available.

Some of the methods mentioned for solving the problems are:

1. Educate and train top management project managers and software developers.
2. Enforce the use of standards, procedures and documentation.
3. Analyze data from prior software projects to determine effective methods.
4. Define objectives in terms of quality desired.
5. Define quality in terms of deliverable.
6. Establish success priority criteria.
7. Allow for contingencies.

8. Develop truthful, accurate cost and schedule estimates that are accepted by management and customer and manage to them.
9. Select project managers based on ability to manage software projects rather than on technical ability or availability.
lO.Make specific work assignments to software developers and apply job performance standards.

The important factor that causes these problems is that the software engineers confine themselves to the technical part of any software product. They don't contribute much to the managerial activities. On the other hand professional managers do not have the technical knowledge.

The programmers and managers tend to have different perception of the problem placed by the customer. So they won't be able to understand one another correctly.

Planning a Software Project:

Managerial activities are as important as the technical activities for the success of a software product. Managers control the resources and the environment in which technical activities occur. Managers also have the responsibility for ensuring that software products are delivered on time and within cost estimates and that products exhibit the functional and quality attributes desired by the customer. Other management responsibilities include developing business plans, recruiting the customers, developing marketing strategies and recruiting and training employees.

Some of the management problems that are to be solved are listed below:
1. Planningfor software engineering projects.
2. Procedures and techniques for the selection of project managers are poor.
3. The accountability of many software engineering projects is poor leaving some question as to who is responsible for various project functions.
4. The ability to accurately estimate the resources required to accomplish a software development project is poor.
5. Success criterion for software development projects are frequently inappropriate. This results in software products that are unreliable difficult to use and difficult to maintain.
6. Decision rules to aid in selecting the proper organizational structure are not available.
1. Decision rules to aid in selecting the correct management techniques for .software engineering projects are not available.
8. Procedures methods and techniques for designing a project control system that will enable project managers to successful control their project are not readily available.
9. Procedures, techniques, strategies and aids that will provide visibility of progress to the project manager are not available.
10. Standards and techniques for measuring the quality of performance and the quality of production expected from programmers and data processing analysts are not available.

Some of the methods mentioned for solving the problems are:
1. Educate and train top management project managers and software developers.
2. Enforce the use of standards, procedures and documentation.
3. Analyze data from prior software projects to determine effective methods.
4. Define objectives in terms of quality desired.
5. Define quality in terms of deliverable.
6. Establish success priority criteria.
7. Allow for contingencies.

8. Develop truthful, accurate cost and schedule estimates that are accepted by management and customer and manage to them.
9. Select project managers based on ability to manage software projects rather than on technical ability or availability.
lO.Make specific work assignments to software developers and apply job performance standards.

The important factor that causes these problems is that the software engineers confine themselves to the technical part of any software product. They don't contribute much to the managerial activities. On the other hand professional managers do not have the technical knowledge.

The programmers and managers tend to have different perception of the problem placed by the customer. So they won't be able to understand one another correctly.

Defining the problem

1. Develop a definitive statement of the problem to be solved, Include a description of the present situation, problem constraints and a statement of the goals to be achieved. The problem statement should be phrase in the customer's terminology.
2. Justify a computerized solution strategy for the problem.
3. Identify the functions to be provided by, and the constraints on, the hardware subsystem, the software subsystem and the people subsystem.
4. Determine system level goals and requirements for the development process and the work products.
5. Establish high-level acceptance criteria for the system.

The first step in planing a software project is to prepare a concise statement of the problem to be solved and the constraints that exists for its solution. The definitive problem statement should written in customer's terminology and should include a description of the present situation and the goals to be achieved by the new system.

Problem definition requires a thorough understanding of the problem domain and the problem environment. Techniques for gaining this knowledge include customer interviews, observation of problem tasks and actual performance of the task by the planner. The planner must be highly skilled in the techniques of problem definition because different customer representatives will have different viewpoints, biases and prejudices that will influence their perception of the problem area. In addition, customer representatives may not be familiar with the capabilities that a computer can offer in their situation and customer representatives are seldom able to formulate their problems in a manner that yields to logical, algorithmic analysis.

The second step in planning a software project is to determine the appropriateness of a computerized solution. In addition to being cost-effective, a computerized system must be socially and politically acceptable. To be cost-effective, a new software product must provide the same services and information as the old system using less time and personnel or it must provide services and information that were impractical without the new system.

Having determined, at least in preliminary fashion, that a computerized solution to the problem, is appropriate, attention is focused on the role to be played by the major subsystems of the computing system. A computing system consists of people subsystem, hardware subsystem and people subsystem plus the interconnections among subsystems.

The functions to be performed by each major subsystem must be identified, the interactions among subsystems must be established and developmental and operational constraints must be detained for each major subsystem Constraints specify numbers and types of equipment, numbers and skill levels of personnel and software characteristics such as performance, accuracy and level of reliability.

2.1.1. Goals and Requirements

Given a concise statement of the problem and an indication of the constraints that exist for its solution, preliminary goals and requirements can be formulated. Goals and targets for achievements and serve to establish the framework for a software development project. Goals apply to both the development process and the work products and goals can be either qualitative or quantitative.

A qualitative process goal the development process should enhance the professional
skills of quality assurance personnel.

A quantitative process goal the system should be delivered within 12 months.
A qualitative product goal the system should make users' job more interesting.
A quantitative product goal the system should reduce the cost of a transaction by 25 percent.

Some goals apply to every project and every product. For instance, every software product should be useful, reliable and cost-effective. Every development process should deliver work products on time and without cost overruns and should provide personnel with the opportunity to learn new skills. Other goals, such as transportability, early delivery of subset capabilities and ease of use by nonprogrammers, depend on the particular situation.
Requirements specify capabilities that a system must provide in order to solve a problem. Requirements include functional requirements, performance requirements and requirements for the hardware, firmware, software and user interfaces. Requirements may also specify development standards and quality assurance standards for both project and product. Requirements should be quantified whenever possible. Quantified requirements such as
1. Response to external interrupts shall be 0.25 second maximum
2. System shall reside in 50K bytes of primary memory excluding file buffers can be used as the basis for acceptance testing of the delivered system. Qualitative requirements such as
1. System small provide real-time response.
2. System shall make efficient use of primary memory are often meaningless and can result in misunderstandings and disagreements between developers and customers. It is difficult to quantify requirements in the planning phase because usually it is not clear what is needed to solve the problem or what can be achieved within the solution constraints.

So, it is needed to formulate meaningful requirements and to state methods that will be used to verify each requirement.

High-level goals and requirements can often be expressed in terms of quality attributes that the system should possess. These high level quality attributes can in turn be expressed in terms of

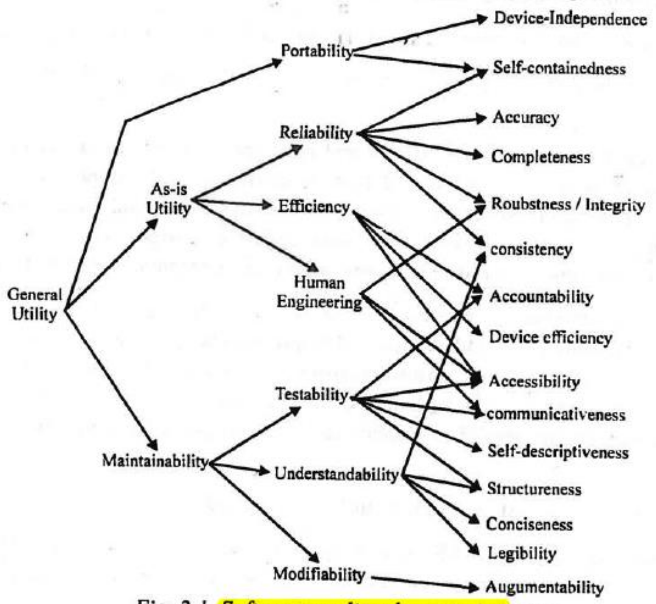


Fig. 2.1. Software quality characteristic tree

attributes that can be built into the work products. For example Understandability can be expressed in terms of consistency, structuredness, conciseness and legibility. Each of these terms can defined in terms of more specific attributes of the source code.
A decomposition of quality attributes into source-code characteristics^ illustrated in Fig.2.1 Table 2.2 provides definitions for some of the terms used in Fig. 2.1

QUALITY ATTRIBUTES

Portability : The case with which software can be transferred from one computer system, or environment to another.
Reliability : The ability of a program to perform a required function under stated conditions for a stated period of time.
Efficiency : The extend to which software performs its intended functions with a minimum consumption of computing resources.
Accuracy : 1) A qualitative assessment of freedom from error. 2) A qualitative measure of the magnitude of error, preferably expressed as a function of the relative error.
Error : A discrepancy between a computed value or condition and the true.
specified or theoritically correct value or condition.
Robustness : The extend to which software can continue to operate correctly despite the introduction of invalid inputs.
Correctness : 1. The extent to which software is free from design defects and from coding defects; that is fault-free 2. The extent to which software meets its specified requirements. 3. The extent to which software meets users expectations.

TABLE 2.3 SOME FACTORS TO CONSIDER IN PROJECT PLANNED

1. Estimation techniques to be used; accuracy required.
2. Life-cycle model, control functions and reviews
3. Organizational structure
4. Level of formality in specification, test plans etc.
5. Level of verification and validation
6. Level of configuration management required.
7. Level of quality assurance required
8. Follow-on maintenance responsibilities
9. Tools to be developed and used
10. Personnel recruitment and training

FACTORS TO CONSIDER IN SETTING PROJECT GOALS

1. New capabilities to be provided
2. Old capabilities to be preserved/enhanced
3. Level of user sophistication
4. -Efficiency requirements
5. Reliability requirements
6. Likely modifications
7. Early subsets and implementation priorities
8. Portability requirements
9. Securely concerns

Developing a solution strategy

1. Outline several solution strategies, without regard for constraints.
2. Conduct a feasibility study for each strategy.
3. Recommend a solution strategy, indicating why other strategies were rejected.

4. Develop a list of priorities for product characteristics.

For any problem, there are always more than one solution. The human tendency will try adopt the first solution that occurs to us. This leads to problems in software engineering. One way of avoiding this problem is to develop a solution strategy. A solution strategy is not a detailed solution plan, but rather a general statement concerning the nature of possible solutions. Strategy factors include considerations such as batch or time -sharing; database or file system; graphics or text; and real-time or off-line processing. A solution strategy should account for all external factors that are visible to the product users and a strategy should be phrased to permit alternative approaches to product design.

Several strategies should be considered before one is adopted. Among these one or more solution strategies must be chosen by the planners to perform feasibility studies and to prepare preliminary cost estimates. The selected strategy provide a framework for design and implementation of the software product.

Solution strategies should be generated without regard for feasibility because it is not possible for humans to be both creative and critical at the same time. All trivial solutions are all enumerated in older to make the best solution strategy. Because the best strategy is a composite of ideas from several different approaches.

The feasibility of each proposed solution strategy can be established by examining solution'constraints. Constraints prescribe the boundaries of the solution space; feasibility analysis determines whether a proposed-strategy is possible within those boundaries. A solution strategy is feasible if the project goals and requirements can be satisfied within the constraints of available time, resources and technology using that strategy. Some iteration and trade-off decisions may be required to bring feasibility and constraints into balance.

Techniques for determining the feasibility of a solution strategy include case studies, worst-case analysis, simulation and construction of prototypes. A prototype differs from a simulation model in that a prototype incorporates some components of the actual system. Prototype implementation usually have limited functionality, low reliability and poor performance characteristics. Prototypes are constructed during the planning phase to examine technical issues and to simulate user displays, report formats and dialogues.

When recommending a solution strategy, it is extremely important to document the reasons for rejecting other strategies. This provides justification for the recommended strategy and may prevent ill-considered regressions at some later date.

A solution strategy should include a priority list of product features. Due to inconsistencies or time and cost overruns at some later time in the development cycle .it may be necessary to postpone or elements some system capabilities. At that time, it is essential that high-level guidance be available to indicate the priorities of essential features, less important features and "nice if features. Without this guidance, a designer or programmer may make serious mistakes in judgment, resulting in customer dissatisfaction with the delivered product. Product priorities are also useful to indicate the manner in which capabilities can be developed and phased into an evolving system.

Planning the development process

1. Define a life-cycle model and an organizational structure for the project.
2. Plan the configuration management quality assurance and validation activities.
3. Determine phase-dependent tools, techniques and notations to be used.
4. Establish a preliminary cost estimates for system development.
5. Establish preliminary development schedule.
6. Establish preliminary staffing estimates.

7.. Develop preliminary estimates of the computing resources required to operate and maintain the system.
8. Prepare a glossary of terms.
9. Identify sources of information, and refer to them throughout the project plan.

planning the development process

illustrated in Table 2.1. The first consideration is to define a product life-cycle model. The software life cycle includes all activities that are required to define, develop, test. deliver, operate and maintain a software product. Different life-cycle models emphasize different aspects of the life cycle. No single life-cycle model is appropriate for all software products. As the model provides a basis for categorizing and controlling the various activities required to develop and maintain a software product it is important to define a life-cycle model lor each software project. If all the members who are involved in a particular software project accept a life-cycle model for it then it improves the project communication, enhances project manageability, resource allocation cost control and product quality life-cycle models discussed here include the phased model, the cost model, the prototype model and the successive versions model.

2.3.1. The**phased life-cycle model**

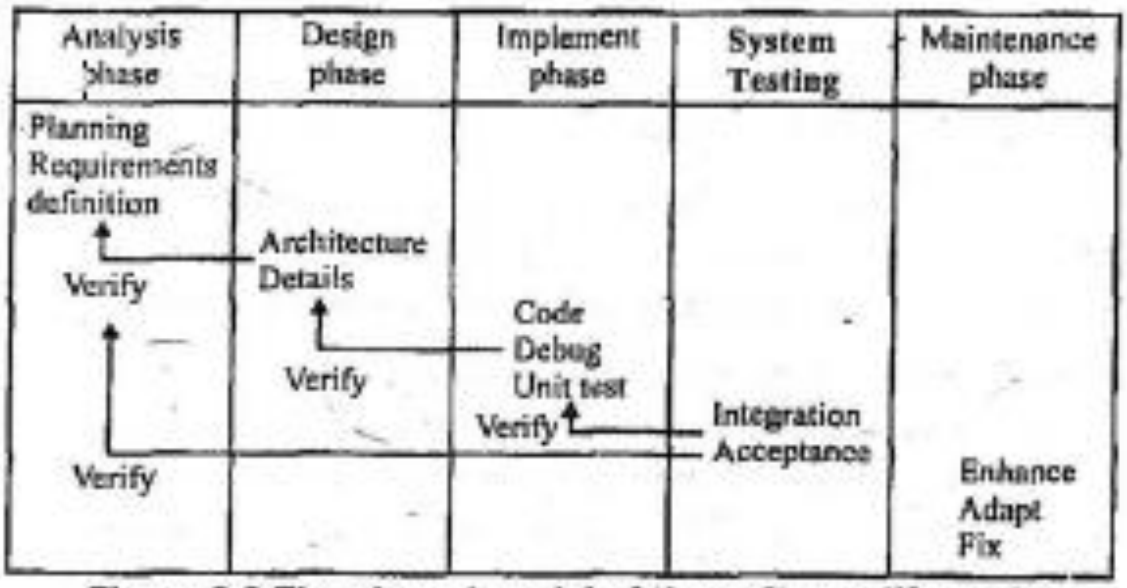


Figure 2.2 The phased model of the software life cycle

In the phased model the software life cycle is segmented into a series of successive activities. In this text, we consider the phased model to consist of the following phases :
analysis, design, implementation, system testing and maintenance. Each phase requires well-defined input information, utilizes well-defined processes and results in well defined products. This model is sometimes called a "waterfall chart" as the products cascade from one level to the next in smooth progression. The basic phased model is presented in fig. 2.2.

Analysis consists of two subphases
1. Planning
2. Requirements definition

The major activities done during planning are summarized in Table 2.1. The products of planning are
1. System Definition & 2. Project plan
The System Definition is typically expressed in English or some other natural language and may incorporate charts, figures, graphs, tables arid equations of various binds, The exact notations used in system definition are highly dependent on the problem area. The format of a system definition is illustrated in Table 2.5.

Table 2.5 Format of a system definition

Section 1 Problem definition
Section 2 System justification
Section 3 Goals for the system and the project
Section 4 Constraints on the system and the project
Section 5 Functions to be provided (hardware/software/people)
Section 6 User characteristics
Section 7 Development / operating / maintenance environments
Section 8 Solution strategy
Section 9 Priorities for system features
Section 10 System Acceptance criteria
Section 11 Sources of information
Section 12 Glossary of term?

e project the preliminary development schedule, preliminary cost and resource estimates, preliminary staffing requirements, tools and techniques to be used and standard practices to be followed. The format of a project plan is presented in Table 2.6
Table 2.6 Format of a project plan
Life-cycle model

Section 1 Terminology / milestone / work products
Organizational structure structure/statements of work.
Section 2 Management structure/team Structure/Work breakdown
Section 3 :Preliminary staffing and requirements staffing and resource schedule.
Section 4 :Preliminary development schedule PERT network / Gantt charts
Section 5 :Preliminary cost estimate
Section 6 :Project monitoring and control mechanisms.
Section 7 :Tools and techniques to be used
Section 8 :Programming languages
Section 9 :Testing requirements
Section 10 Supporting documents required
Section 11 Manner of demonstration and delivery
Section 12 Training schedule and materials
Section 13 Installation Plan
Section 14 : Maintenance considerations
Section 15 : Method and time of delivery
Section 16 : Method and time of payment
Section 17 : Sources of information

Requirements definition is concerned with identifying the basic functions of the software components in a hardware/software people system. The product of requirements definition is a specification that describes Processing environment the required software functions performance constraints on the software (size, speed machine configuration) Exception handling Implementation priorities).probable changes and likely modifications and the acceptance criteria for the software.

In this phase, emphasis is placed on what the software is to do and the constraints under which it will perform its functions.

Software design phase follows the analysis phase. Design phase consists of
1. Architectural design
2. Detailed design

Architectural design involves the identification of the software components, by decoupling and decompiling the software components. Detailed design is concerned with the details of how to package the processing modules and how to implement the processing algorithms, data structures and interconnections among modules and data structures. Detailed design involves adaptation of existing code, modification of standard algorithms, invention of new algorithms, design of data representations and packaging of the software product.

Implementation phase follows the design phase. This phase consists of the following subphases.
1. Code 2. Debug 3. Unit test

Coding involves the translation of design specifications into source code Modern programming languages provide many features to enhance the quality of the source code.

Errors discovered during the implementation phase may include errors in the data interfaces between routines, logical errors in the algorithms, error in data structure layout and failure to account for various processing cases. The source code may contain errors that reflect the failure to capture the customer's requirements, design errors that reflect failure to translate requirements into design specifications and implementation errors that reflect failure to correctly translate design specifications into source code.

Unit test involves the testing of each and every processing module whether it works correctly for the given test data.

After writing the source code (i.e. after the implementation phase), the software product is to be tested.

Testing involves the following two activities.
1. Integration testing
2. Acceptance testing

The various modules of the software system would have been developed by different groups of programmers. The unit test performed on the processing modules ensures that the modules will perform the required function as it stands alone. In order for the whole software system to function correctly, the modules are to be integrated. Careful planning is needed to improve the availability of the modules for the integration when needed.

Acceptance testing involves planning and execution of various types of tests in order to demonstrate that the implemented software system satisfies the requirements state in the requirements document.

The software system is released for the production work after being accepted by the customer and it enters the maintenance phase of the following activities.
1. Enhancement 2. Adaptation 3. Fix IJ

Enhancement involves the improvement of the capabilities of the software system. Adaptation is making the software system suitable for new processing environments. Fixing involves the correcting of the software bugs.

The phased model of the software is very simple. There are no milestones in the model, no mention of the documents generated or the various reviews conducted during the development process. The phased life cycle model is valid only if a reasonably complete set of specification for the software product can be written at the beginning of the life cycle.

2.3.2. Milestones; Documents and Reviews

A life cycle model which emphasizes the milestones, documents and reviews is illustrated in figure 2.3".

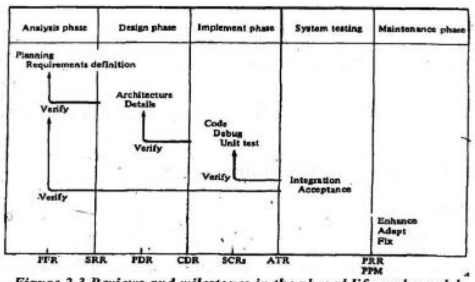


Figure 2.3 Reviews and milestones in the phased life-cycle model

REVIEW
Product Feasibility Review

SRR : Software Requirements Review

PDR : Preliminary Design Review
Detailed Design Specification SCR : Source Code Review
ATR : Acceptance Test Review PRR : Product Release Review PPM : Project post-mortem

WORK PRODUCTS REVIEWED
System Definition
Project plan
Software Requirement Specification
preliminary user's manual
preliminary Verification plan
Architectural Design Document
User's manual
Software Verification plan a Walkthroughs & Inspections of the Source Code
Acceptance Test plan
All of the Above
Project Legacy

To improve product visibility establishing milestones, review points and standardized documents are" very ijginortarit. These help the project team members-to-assess the progress of the project. During the Hfe cycle of the product various documents are generated. It may be needed to review these documents to improve the feasibility of the product
1. A system Definition and Project plan are prepared using the formats of Tables 2.5 and 2.6 Product Feasibility Review is then held to determine the feasibility of the project continuation. The outcome of the review may be the termination of the project, redirection of the project or the continuation of the project. Redirection may mean changes in the "system definition and project plan which requires another product feasibility review to be held after the changes are made.
2. A software requirement specification is prepared. This clearly defines each essential requirement for the software product as well as the external interfaces to hardware, firmware, other software and people. The format of a software requirement specification is»illustrated in Table 2.7
Table 2.7 Format of a Software Requirement Specification

Section 1 :product overview and summary
Section 2 :Development / operations / Maintenance environments
Section 3 :External interfaces and data flow.
Section 4 :User displays / report formats /User Command summary
Section 5 :High-level data flow diagrams /Logical data sources /. sinks Logical data stores Logical datadictionary
Section 6 :Functional specifications /Performance requirements

Section 7 :Exception conditions / exception handling
Section 8 :Early subsets implementation priorities /Foreseeable modifications and enhancements
Section 9 :Acceptance criteria /Functional and performance test Documentation standards
Section 10 : Design guidelines (hints and constraints)
Section 11 : Sources of information
Section 12 : Glossary of terms

A preliminary version of the User's manual is prepared. It provides a vehicle of communication between customer and developer. It is prepared using the information from the system Definition the outlying of a typical User's Manual is illustrated in table 2.8
Introduction *Table 2.8 Outline of the User's Manual* -Section 1

2.3.3. The Cost Model

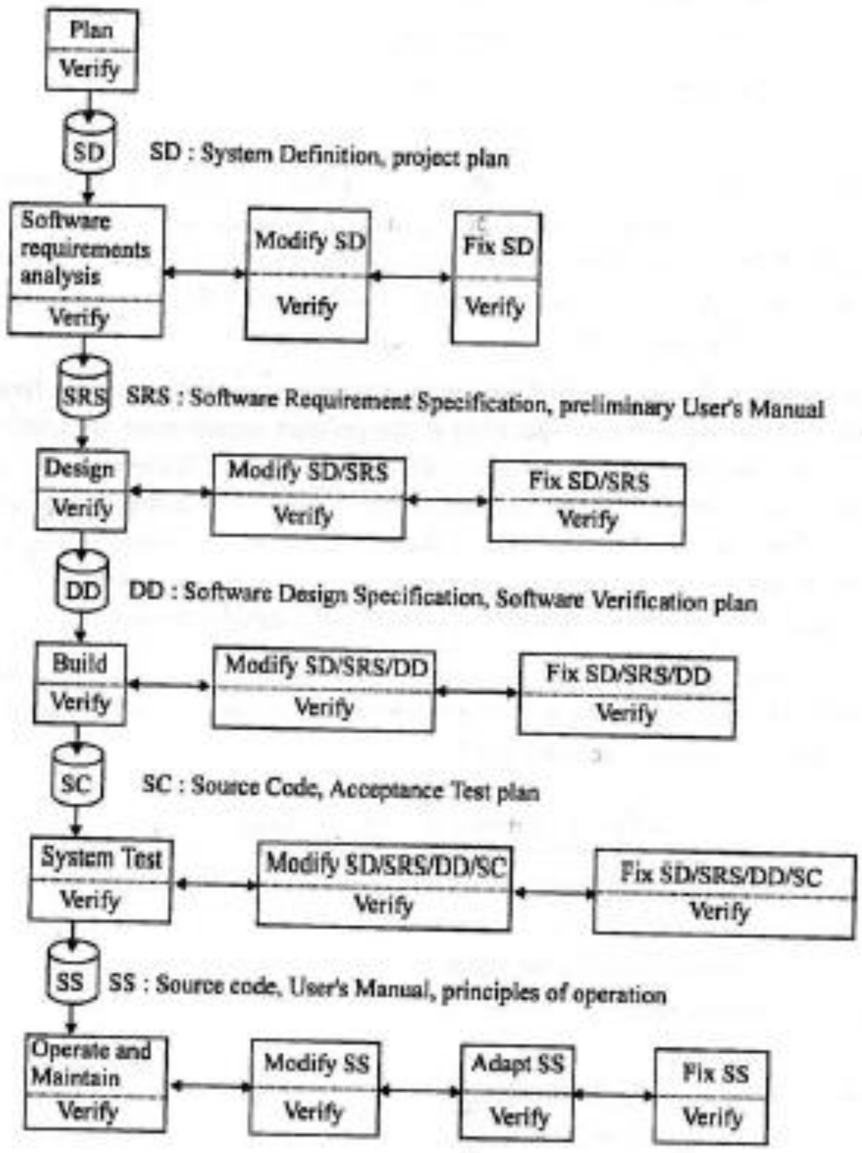

Plan
Verify
SD
SD : System Definition, project plan
Software requirements analysis
Verify
Modify SD
Verify
Fix SD
Verify
SRS
SRS : Software Requirement Specification, preliminary User's Manual
Design
Verify
Modify SD/SRS
Verify
Fix SD/SRS
Verify
DD
DD : Software Design Specification, Software Verification plan
Build
Verify
Modify SD/SRS/DD
Verify
Fix SD/SRS/DD
Verify
SC
SC : Source Code, Acceptance Test plan
System Test
Verify
Modify SD/SRS/DD/SC
Verify
Fix SD/SRS/DD/SC
Verify
SS
SS : Source code, User's Manual, principles of operation
Operate and Maintain
Verify
Modify SS
Verify
Adapt SS
Verify
Fix SS
Verify

Fig. 2.4. The cost model of the software life cycle

Outline of the manual
Section 1: Product overview and rationale/Terminology and basic features/Summary of displays and report formats/
Section 2: Getting Started/Sign-on/Help mode/Sample run
Section 3: Modes of operation/Commands / dialogues / reports
Section 4 : Advanced features

The cost of conducting a software project is the sum of the costs incurred in conducting each phase of the project cost incurred within each phase include the cost of performing the processes and preparing the products for that phase, plus the cost of verifying that the products of the respect to all previous phase.

Modifications and corrections to the products of previous phase are necessary because the processes of the current phase will expose inaccuracies, inconsistencies and incompleteness in those products and because changes in customer requirements, schedules, priorities and budget will dictate modifications.

The cost of producing the system Definition and the Project Plan is the cost of performing the planning function and preparing the documents plus the cost of verifying that the system Definition accurately states the customers needs and the cost of verifying that the Project Plan is feasible.

The cost of preparing Software Requirement Specification includes the cost of performing requirements definition and preparing the specification document, plus, the cost of modifying the correcting the system Definition and the Project Plan plus the cost of verifying that the Software Requirement Specification is complete and consistent with respect to the system Definition and the customers needs.

Similarly the cost of design is the cost of performing the design activities and preparing the design specification and the test plan, plus the cost of modifying and correcting the system Definition, the Project Plan and the Software Requirement Specification, plus the cost of verifying the design against the requirements, the system Definition and the Project Plan.

The cost of product implementation is the cost of implementing, documenting, debugging, and unit testing the source code, plus the cost of completing the user's Manual, the verification plan, the maintenance procedures, the installation and training instructions, plus the cost modifying and correcting the System Definition, the project plan, the Software Requirement Specification, the Design Specification and the Verification Plan, plus the cost of verifying that the implementation is complete, consistent and suitable with respect to-the System Definition, the Software Requirement Specification and the design documents.

The cost of system testing includes the cost of planning and conducting the tests, plus the cost of modifying and correcting the source code and the external documents during system testing, plus the cost of verifying that the tests adequately validate the product.

Finally, the cost of software maintenance is the sum of the costs of performing product enhancements, making adaptations to new processing requirements and fixing bugs. Each of these activities may involve modifying and correcting any or all of the supporting documents and running a large number of test cases to verify the correctness of the modification. The cost of conducting a software project is summarized in fig. 2.4.

Given this view, it is clear that modifications or corrections to the Software Requirement Specification and design documents in subsequent phases of the life cycle are too costly. This is because not only must the documents be modified, but all intermediate work products must also be updated.

2.3.4. The prototype Life-cycle Model
Another view of software development and maintenance is presented in fig. 2.5.

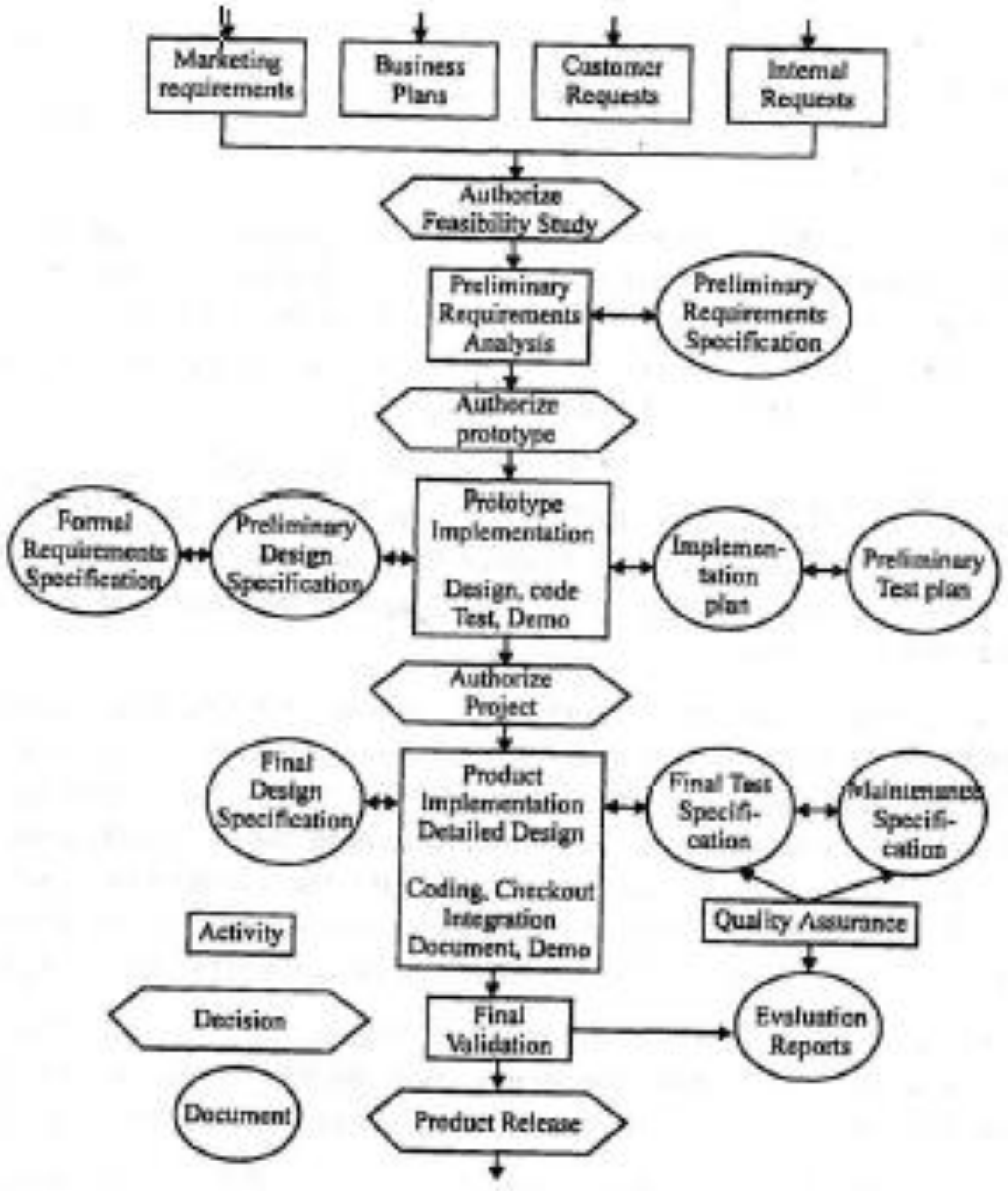


Fig. 2.5

This model emphasizes the sources of product request, the major decision points and the use of prototypes. A prototype is a model of a software product. In contrast to a simulation model, it incorporates components of the actual product. Typically, a prototype exhibits limited functional capabilities, low reliability and / or inefficient performance.

There are several reasons for developing a prototype. One important reason is to illustrate input data formats, messages, report and interactive dialogues for the customer. This helps in explaining various processing options to the customer and to gain better understanding of the customer's needs.

The second reason for implementing a prototype is to explore technical issues in the proposed product.

The phased model is applicable only when it is possible to write a reasonably complete set of specifications for a .software product at the beginning of the life cycle. The third reason for developing a prototype is in situations where phased model is not appropriate.

The nature and extent of prototyping to be performed on a particular software project is dependent on the nature of product. New versions of existing products can be developed using phased life-cycle model with little or no prototyping Development of a totally new product will probably involve some prototyping during the planning and analysis phase or the product may be developed by interacting through a series of successive designs and implementations (viz. successive versions).

2.3.5 Successive versions :

2.3.5 Successive versions :

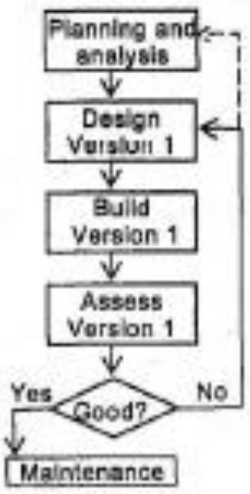


Fig. 2.6(a) Design & Implementation Implementation of successive versions

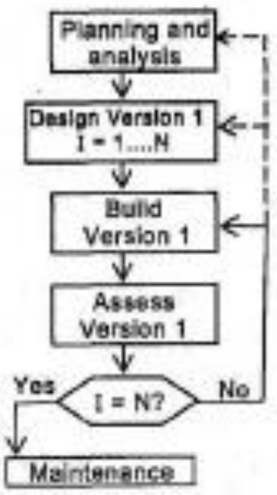


Fig. 2.6(b) Analysis & design followed by implementation of successive versions

The method of successive versions is an extension of prototyping. At each interaction the level of capability is increased. Fig. 2.6a illustrates the design and implementation successive versions.

Fig. 2.6(a) illustrates the design and implementation of successive version where planning and analysis phase is followed by illustrate design, implementation and assessment of successive versions. The dashed line dictates the need for further analysis before designing version 1+1 after the assessment version I.

In fig. 2.6b version I through N are designed prior to any implementation activity. The dashed lines indicate that the implementation of version I may dictate the need for further analysis and design before proceeding with implementation of version 1+1

In reality, the development cycle for a software product is a composite of the various models presented in this action. A particular project may adopt the structure of one particular model, but element of each model are likely to be found in every project. Not all the documents mentioned in this section are developed for all software products. A minimal set of documents for a software product includes a statement of requirements, a design specification, a test plan and a user's manual.

Adopting a product life-cycle model improves software quality, increasing programmer productivity, better management control and improved morale.

Planning an organization structure

The tasks that are performed during the lifetime of a software product include planning, product development, services, publications, quality assurance, support and maintenance. The Planning task identifies external customers and internal product needs, conducts feasibility studies and monitors progress from beginning to end of the product life cycle. The development task specifies, designs, implements, debugs, tests and integrates the product. The services task provides automated tools and performs configuration management, product distribution and miscellaneous administrative support. The publications task develops user's manuals installation instructions, principles of operation and other supporting documents. The quality assurance that provides independent evaluation of source code and publications prior to releasing them to customers. The support task promotes the product, trains users, installs the product and provides continuing liasion between users and other tasks. The maintenance task provides error correction and minor enhancement throughout the productive life of the software product. Majors enhancements and adaptation of the software to new processing environments are treated as new development activities in this scheme. Several variations on this structure are possible Methods for organizing these tasks include the project format, the functional format and the matrix format.

2.4.1. Project Structure

Project format

Use of project format involves assembling a team of programmers who conduct a project from start to finish. Project team members do the product definition, designing, implementation, testing, conduct the project reviews and prepare the supporting documents. After completing thee phases some of the team members will stay with the product during installation and maintenance while others will be moved on to a new project. They too have the responsibility for the maintenance of the delivered product. Project team members typically work on a project for 1 to 3 years and are assigned to new projects on completing of the current one.

Functional format

In the project format project team members conduct the project from the beginning to end. But in functional format a different team of programmers performs each phase of the project. The team members performing one particular phase generate the work products for that phase. It is then passed on to the next team who in turn generate the work products for. the phase.

The various teams along with the function performed by them and the work products generated by them are given in the following table.

Team Function and / or **the work products**

1. Planning and Analysis team System Definition Project Plan
2. Product Definition team Performs software requirements analysis prepares Software Requirement Specification
3. Design team designs the product to confirm with the System Definition and Requirement Specification
4. Implementation team implements, debugs and unit tests the product
5. System testing team does the integration testing and acceptance testing.
6. Quality Assurance team certifies the quality of all work products.

7. Maintenance team maintains the product during its useful life.

Planning and analysis team prepares system Definition and project plan. These work products are then passed to the Product Definition team. These work products are passed to the next team in the same way as they are evolved.

A variation on the functional format involves three teams.

1. Analysis team
2. Design and implementation team
3. Testing and maintenance team.

In this scheme, a support group provides publications, maintains the facilities and provides installation and training. The functional format requires more communication among teams than the project format. As the team members are concerned with one particular phase they become specialists which results in proper documentation. Team members are rotated from one function to another to provide career development.

Matrix format

The matrix format is illustrated in fig. 2.7.

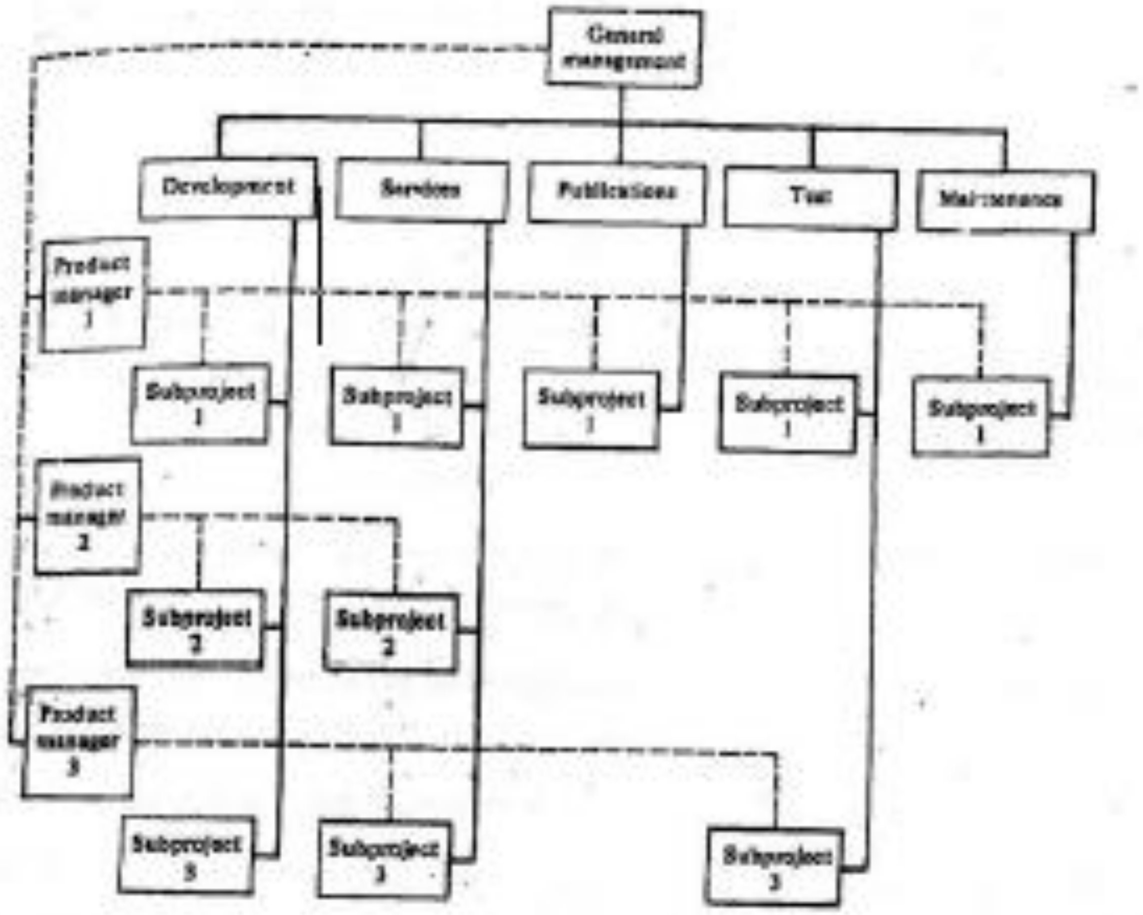


Fig.2.7 Product – function matrix organization

Each function has its own management team and a group of specialists who are concerned only with that function. The organization is having different team of specialist for each of the function of the organisation. The various functions of the organization are
1. Development
2. Services
3. Publications
4. Test
5. Maintenance

The organization may take more than one project at any time. Eacn project has a project manager. The project manager is organizationally a member of the planning function or the development function. The project manager generates and reviews documents and may participate in design implementation and testing of the product.

Software development team members organizationally belong to the development function but work under the supervision of a particular project manager. The team members may work on one or more project under the supervision of one or more project managers.

In matrix format, each member has at least two bosses. This results in ambiguities that are to be resolved. In spite of the problems created by matrix organizations, they are increasingly popular because special expertise can be concentrated in particular function, which results in efficient and effective utilizetion of personnel. Also, project staffing is eased because personnel can be brought onto a project as needed and returned to their functional organization when they are no longer needed. The workload of the team members are balanced so that returning to their functional organization are assigned to other projects or they spend some time in their functional to acquire new skills."

2.4.2. Programming team structure

Every programmer team must have an internal structure. The best structure for a particular project depends on the nature of the project and the product and on the characteristics of the individual team members. There are three basic team structures. They are
1. Democratic teams
2. Chief programmer teams
3. Hierarchical team

Characteristics of the various team structures are discussed below.

Democratic teams

All the team members participate in all decisions. The idealized democratic team structure is also called as an "egoless team". The management and communication paths in an egoless team are illustrated in fig. 2.8

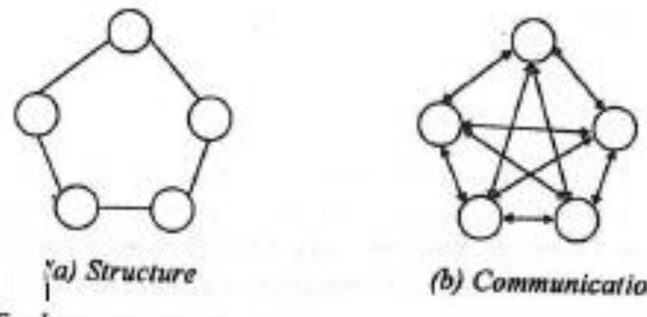


Fig. 2.8 Egoless programming team structure and communication paths

In an egoless team, goals are set and decision are made by group consigns. Groups leadership is moved from one member to the other based on the activity and the capability of the team members.

A democratic team is slightly different from that of the egoless team. In a democratic team one individual is designated as the team leader. He is responsible for the coordination of the team activities and also making decisions in critical situations. The leadership is not rotated among the team members as it happens in the egoless team.

Advantages of the democratic team includes :

1. Opportunity for each team member to contribute to decisions
2. Team members can learn from one another
3. Involvement in the project is increased as the team members are participating in each and every activity of the team.
4. As the problem is discussed in an open, nonthreatening work environment, the job satisfaction is higher in the democratic team.

Despite having so many advantages the democratic team also has some disadvantages.

1. Decision-making in the critical situations are difficult as it requires the agreement of all the team members.
2. Co-ordination among the team members is essential
3. Less individual responsibility and authority can result in less initiative and less personal drive from team members.

Chief Programmer Teams

The management structure and communication paths of chief programmer teams are illustrated in fig. 2.9.

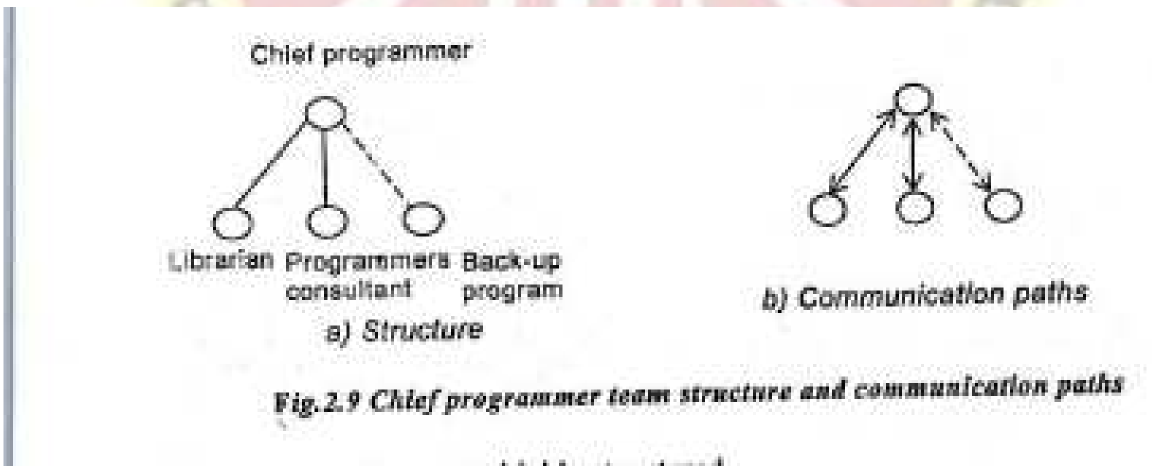


Fig.2.9 Chief programmer team structure and communication paths

Chief programmer teams are highly structured. A team of programmers work under the chief programmer. The chief programmer designs the product, implements critical parts of the products and makes all major technical decisions. Work is allocated to the programmers by the chief programmer. The programmers write code, debug, document and unit test it.

The back-up programmer serves as consultant to the chief programmer on various technical problems; provides liaison with the customer, the publication group and quality assurance group. They may also perform some analysis, design and implementation under supervision of the chief programmer. A program librarian maintains all the program listings, design documents, test plans, etc. in a central location.

The chief programmer is assisted by an administrative program manager, who handles administrative details. In chief programmer team structure, the emphasis is to provide complete technical and administrative support to the chief programmer who has responsibility and authority for development of software product.

Chief programmer teams are effective in two situations, first, in data processing applications where the chief programmer has responsibility for sensitive financial software packages and the packages can be written by relatively unskilled programmers and second, in situations where one senior programmer and several junior programmers are assigned to a project. In the latter case, the chief, the chief programmer team structure is used to train the junior programmer and to evolve the team structure into a hierarchical or democratic team when the junior programmers have obtained enough experience to assume responsibility for various project activities.

Hierarchical team **structure**

The hierarchical team structure occupies a middle position between the extremes of democratic items and chief programmer teams. The management structure and communication paths in a hierarchical team are illustrated in fig. 2.10.

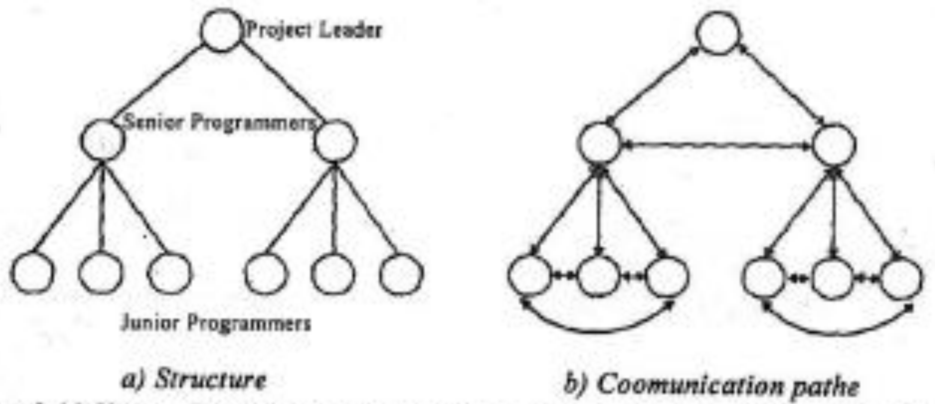


Fig. 2.10 Hierarchical Programming team structure and communication paths

Team structure usually reflects the structure of the software product. If the software product is a hierarchical one then hierarchical team structure is well suited to the development of this product. Each major subsystem in the hierarchy can be assigned to a team. The team leaders will report to the project leader. For effective communication and supervision of work activities, the number of immediate subordinates at each level must be limited to five to seven people.
A large project may have several levels in the hierarchy.

There is one major disadvantage in a hierarchical structure. The competent and experienced programmers may be promoted to management positions. Eventhough higher salary and. prestige are associated with higher positions, the promotion may have doubly negative effect of losing a good programmer and creating a poor manager. The competent programmers need not have a good communication skill and managerial skill that are essential for a good manager.

We have discussed three basic programming team structures: the democratic structure, the chief programmer structure and the hierarchical structure. Variations on these structure are also possible. For example, each member of a democratic team might be the project leader of a hierarchical team that is responsible for a subsystem of the software product.

2.4.3. Management by objectives

Depending upon the nature of the software project, project format and team structure are identified. After identifying the project format and team structure, the project leader has an important responsibility of controlling the various activities of the team members. It is necessary for a project leader to establish methods for directing and controlling the activities of individuals assigned to the project. An effective technique is to have written job description for each project leader. At the beginning of the project, project members develop their understanding of the project and their role in the project. These job descriptions are modified in consultation with the project leader. This reduces the misunderstandings and false assumptions of both manager and managee.

Management by objectives (MBO) is a related technique. Using MBO, the project members set their own goals and objectives with the help of their supervisor, participate in settling of their supervisor's goal and they are evaluated. The highly skilled and soft-motivated software engineers treat MBO as an effective technique.

Management by objectives can be applied at all levels of an organization, and can include product oriented objective, such as completion of design or testing and self-improvement objectives such as skills to acquire and preparations for advancement.

Eventhough MBO is an increasingly popular technique, it is sometimes criticized as a cumber some bureaucratic exercise. This criticism is justified not because the concept of MBO is faulty, but because "flie local implementation of MBO is often incorrect. Typical problems with MBO are

1. Employees are required to state excessive number of objectives (20 or 30)
2. The reporting period is too long (6 months or more)

Objectives of software projects should be specific, few in number and achievable in a short time period (1 to 3 months) Furthermore objectives should be phrased so that attainment of an objectives can be readily determined.

Other planning activities.

These include

1. Planning the configuration management and quality assurance functions.
2. Planning for independent verification and validation
3. Planning phase - dependent tools and techniques

2.5.1. Planning for configuration management and Quality Assurance

Configuration management is concerned with controlling changes in work products, accounting for the status of work products and maintaining the program support library. Quality assurance develops and monitors adherence to project standards, programs audits of the processes and work products and develops and performs the acceptance tests.

During the planning phase, the configuration management and quality assurance procedures to be used are specified and the tools needed to perform configuration management and quality assurance are identified and acquired

During the design phase, configuration management and quality assurance of the requirements and design specifications are performed, adherence to project standards is monitored and the tools needed to perform configuration management and quality assurance are used. During the implementation and testing phases, configuration management and quality assurance of requirements, design specifications and source code are performed. During the testing phase, acceptance testing and preparation of test results are performed. During the planning phase, these activities should be planned and adequate resources should be budgeted to permit successful configuration management and quality assurance.

2.5.2. Planning for Independent Verification and Validation

On some critical software projects, an independent organization may provide verification and validation of work products.

Independent organization performing the verification and validation results in high quality software products. But the cost of performing independent verification and validation may be as much as 25 percent of the development cost. Verification ensures that various work products are complete and consistent with respect to other work products and customer needs. Thus, an external organization might verify that the design specifications are complete and consistent with respect to the System Definition and the software product specifications and that the source code is complete and consistent with respect to the design specifications and the requirements. Validation is concerned with assessing the quality of a software system in its actual operating environment. Validation typically involves Planning and execution of test cases. On projects using independent verification and validation, test cases are developed and executed by the independent organization.

2.5.3. Planning Phase-Dependent Tools and Techniques

Automated tools, specialized notations and modern techniques are often used to develop software requirement specifications architectural and detailed designs and the source code. Automated testing tools may be used for unit testing system testing and acceptance testing Management tools such as PERT charts, Gantt charts work breakdown structures and personnel staffing charts may be used to tract and control progress.

2.5.4.Other Planning Activities

Other planning activities include preparing preliminary cost estimates for product development establishing a preliminary development schedule, establishing preliminary staffing levels and developing preliminary estimates of the computing resources and personnel required to operate and maintain the system.

Unit-2:

Software Cost Estimation:

Estimating the cost of a software is one of the difficult task in software engineering. During the planing phase, it is very difficult to make an accurate estimation of the software as there are a large number of unknown factors at that time. This difficulty in making an accurate cost estimation during the planning phase coupled with the competitive nature of business usually results in cost and schedule overruns of software products.

In order to overcome the problem of cost overrun, some organizations use a series of cost estimates. A preliminary estimate is prepared during the planning phase and presented at the project feasibility review.

An improved estimate is prepared and presented at the software requirements, review and the final estimate is presented at the preliminary design review. Each estimate is a refinement of the previous one and is based on the additional information gained as a result of additional work activities. For some software project there may be several product options. Sometimes these product options along withthe costs are presented at the reviews. This allows the customer to choose a cost effective solution from a range of possible solutions.

Customers to have a clear picture of the cost estimates and schedule sometimes fund the analysis phase and the preliminary design phase on separate contracts. Contracts are sometimes awarded to multiple software development organization for analysis and preliminary design phases alone by the customer. The customer then choose an organization to develop the software product on the basis of the analysis and preliminary design competitions.

Major factors that influence software costs are listed in table 3.1.

Table 3.1. Major factors that influence software cost.

Programmer ability Product size Available time Required reliability Level of technology.

Software Cost factors

There are many factors that influence the cost of the software. The effects of most of these factors and hence the cost of a development or maintenance effort are difficult to estimate.

3.1.1. Programmer Ability

The individual attributes of project and their familiarity with the application area is an important software cost factor. On very large products, the differences in individual programmer ability will tend to average out, but on projects utilizing five or fewer programmers, individual differences in ability can be significant.

3.1.2. Project Complexity

Generally, there are three categories of software products: application programs, utility programs and system level programs.

Application programs which include data processing and scientific programs are developed in the environment provided by the language compiler. Interactions with the operating system are limited to job control statements and real-time support facilities provided by the language processor. Utility programs such as compilers, linkage editors and inventory systems are written to provide user processing environments, Utility programs make sophisticated use of operating system facilities. System programs such as data base management system operating and real-time systems interact directly with the hardware.

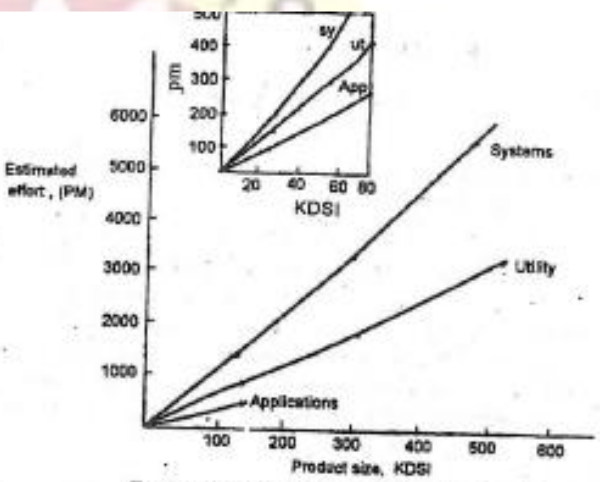


Fig.3.1 COCOMO effort estimates (BOE81)

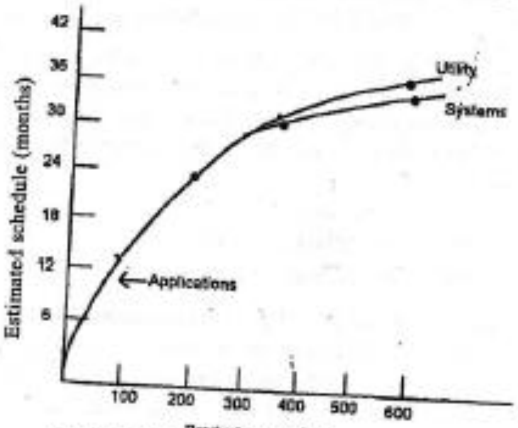


Fig.3.2 COCOMO schedule estimates

Brooks states that utility programs are three time as difficult to write as application programs and that systems programs are three times as difficult to write as utility programs. His levels of product complexity are thus 1-3-9 for applications utility systems programs.

Boolean provides equations to predict total programmer-months of effort. PM in terms of the number of thousands of delivered source instructions. KDSI Programmer cost for a software project can be obtained by multiplying the effort in programmer-months by the cost per programmer-month. The equations were derived by examining historical data from a large number of actual projects.

Application Programs	: PM = 2.4 * (KDSI) * * 1.05
Utility Programs	: PM = 3.0 * (KDSI) * * 1.12

Systems programs : PM = 3.6 * (KDSI) * * 1.20

Graphs of these equations are presented in fig 3.1. These equations state that roughly twice as much effort is required to develop a utility program and roughly three times as much effort is required to develop a system program, as is required to develop an application program of 60K lines. It can be observed in fig,3,1, that the ratios grow with larger programs.

The development time for a program , as given by boehm, is

Application programs	: TDEV = 2.5 * (PM) * * 0.38
Utility programs	: TDEV = *2%* * (PM) * * 0.35
Systems programs	: TDEV = 2.5 * (PM) * * 0.32

Given the total programmer-months for a project and the nominal development time required, the average staffing level can be obtained by a simple division. For 60 KDSI program, we obtain the following results.

Application programs	: 176.6 PM/17.85 MO = 9.9 Programmers.
Utility programs	: 294 PM/18.3 MO = 16 Programmers.
Systems programs	: 489.6 PM/18.1 MO = 27 Programmers.

These figures represent average staffing levels. During analysis and architectural design, only a few people are required and about 125 to 150 percent of average staffing will be required during implementation phase.

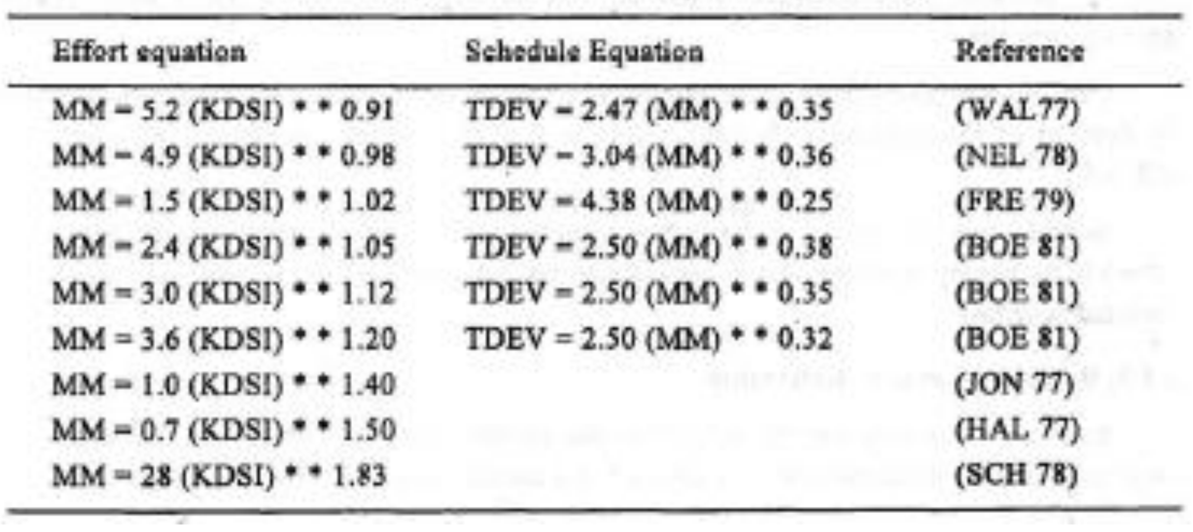
These results are presented only for the purpose of illustration. These equations should not be used to estimate software effort without understanding the assumptions and limitations of the method. The independent variable in these equations is delivered source instructions. The estimates are thus no better than our ability to estimate the final number of instructions in the program. This is a difficult estimate to make in the planning phase.

3.1.3. Product size

Obviously a large software product is more expensive to develop than a small one. Boehm's equations indicate that the rate of increase in required effort grows with the number of source instructions at an exponential rate slightly greater than 1. Several different effort and development schedule estimators are presented in Table 3.2

From table 3.2, it can be noted that the development time estimators are in close agreement than the effort estimators,

Table 3.2 Effort and schedule estimators (BOE 81)



Effort equation	Schedule Equation	Reference
MM = 5.2 (KDSI) * * 0.91	TDEV = 2.47 (MM) * * 0.35	(WAL77)
MM = 4.9 (KDSI) * * 0.98	TDEV = 3.04 (MM) * * 0.36	(NEL 78)
MM = 1.5 (KDSI) * * 1.02	TDEV = 4.38 (MM) * * 0.25	(FRE 79)
MM = 2.4 (KDSI) * * 1.05	TDEV = 2.50 (MM) * * 0.38	(BOE 81)
MM = 3.0 (KDSI) * * 1.12	TDEV = 2.50 (MM) * * 0.35	(BOE 81)
MM = 3.6 (KDSI) * * 1.20	TDEV = 2.50 (MM) * * 0.32	(BOE 81)
MM = 1.0 (KDSI) * * 1.40		(JON 77)
MM = 0.7 (KDSI) * * 1.50		(HAL 77)
MM = 28 (KDSI) * * 1.83		(SCH 78)

3.1.4. Available time

Total project effort is sensitive to the calendar time available for project completion. We may think that the total effort required .to complete the project decreases with the increase in the development time for that project, But that is not true. If the development time for the project is expanded beyond some limit it will increase the total effort required to develop the software project. Several investigators have studied the question of optimal development time. The results of these studies are summarized in fig.3.3.

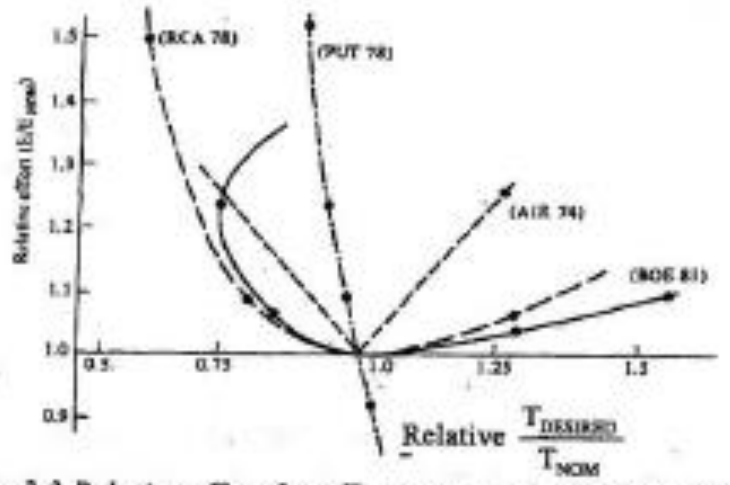


Fig 3.3 Relative effort for off non terminal schedules (BOE 81)

It can be noted that all the curves other than putnam curve are in closer agreement in fig 3.3. According, to putnam, project is inversely proportional to the fourth power of development time. E = K/(Td * * 4). If the development time is compressed, this curve indicates an extreme penalty and if the development time is expanded, it indicates an extreme reward. This formula is not correct as it predicts zero effort for infinite development time.

Putnam also states that the development schedule^0 cannot be compressed below about 86 percent of the nominal schedule, regardless of the number of people or resources utilized.

Boehm states, "There is a limit beyond which a software project cannot reduce its schedule by buying more personnel and equipment. This limit occurs roughly at 75% of the nominal schedule"

3.1.5. Required Level of Reliability

Software reliability can be defined as the probability that a program will perform a required function under stated conditions for a stated period of time. Reliability can beexpressed in terms of accuracy, robustness, completeness and consistency of the source code. To ensure higher reliability, there is a cost associated with the increased level of analysis, design, implementation and verification and validation

Table 3.3. Development effort multipliers for software reliability (BOE 81)

Category	Effect of failure	Effort multiplier
Very low	Slight inconvenience	0.75
Low	Lossed easily recovered	0.88
Nominal	Moderately difficult to recover losses	1.00
High	High financial loss	1.15
Very high	Risk to human life	1.40

In a software project, some product failure may cause little inconvenience to the user, while failure of other products may lead to high financial loss. So the required level of reliability should be established during the planning phase by considering the cost of software failures. Boehm's reliability categories and the effort multiplier for each category are presented in Table 3.3.

The multipliers range from 0.75 for very low reliability to 1.4 for very high reliability. The effort ratio is thus 1.87 (1.4/0.75).

3.1.6. Level of Technology

The level of technology in a software development project is reflected by the programing language, the abstract machine (hardware and software), the programming practices ;uu the software tools used. As the program statements written in high level language instead of assembly language increases programmer productivity by a factor of 5 to 10. Modern languages such as Ada provide additional features to improve programmer productivity and software reliability.

These features include strong type-checking, data abstraction, separate compilation, exception handling, interrupt handling and concurrency mechanisms. The abstract machine is the set of hardware and software facilities used during the development process. Productivity will suffer if programmers must learn a new machine environments as part of the development process, or if the machine is being developed in paralled with the software or if programmers have only restricted access to the machine. Familiarity with the stability of and ease of access to the abstract machine all influence programmer productivity and hence the cost of a software project.

Modern programming practices and modern development tools will reduce the development effort. Modern programming in practices include use of systematic analysis ad design techniques, structured design notations, walkthroughs and inspections, structured coding, systematic testing and a program development library. Software tools range from elementary tools, such as assemblers and basic debugging aids, to compilers and linkage editors, to interactive text editors and data base management environments that include configuration management and automated verification tools.

Software cost estimation techniques

Within most organizations, software cost estimates are based on past performance. The cost and productivity data must be collected on current project in order to estimate. future ones. Cost estimates can be made either top-down or bottom-up. Top-down estimation first focuses on system-level costs, such as the computing resources and personnel required to develop the system, as well as the costs of configuration and management, quality assurance, system integration, training and publications. The two top-down cost estimation technique which we are going to discuss..

1. Expert Judgment
2. Delphi cost Estimation

Bottom-up cost estimation first estimates the cost to develop each module or subsystem. Those cost are combined to arrive at an overall estimate, Work Breakdown structures and Algorithmic cost estimators are the bottom-up cost estimators.

Top-down estimation has the advantage of focusing on system level costs, but may overlook various technical factors in some of the modules to the developed. Bottom-up estimation emphasizes the costs associated with developing individual system components, but may.fail to account for system-level costs, such as configuration management and quality control. Generally, both top-down and bottom-up estimates are developed, compared and iterated to eliminate differences,

3.2.1. Expert Judgment

The most widely used cost estimation technique is expert judgment. It is an inherently top-down estimation technique. Expert judgment relies on the experience, background and business sense of one or more key people in the organization.

An expert on identifying the similarity between the new project and project they have developed in the recent past, estimates the cost of the new project in the following manner. If the new system has similar functions as that of the old one and also has some more functions, then it increase the time and cost estimates of the new project. As the group of programmers are going to work in the same environment and many of the same people who have developed the previous system are available to develop the new system we can reduce our cost estimate based on these information. Based on the cost and schedule of the previous project and analyzing the new project, the expert estimates the cost and schedule of the new project.

This cost estimation techniques has many disadvantages. The expert may be confident that the project is similar to a previous one, but may have overlooked some factors that make the new project significantly different. The other disadvantage of this technique is that the expert may not be having experience with the project similar to the present one in which case cost estimation of the new project is a difficult -task to overcome these disadvantages, groups of experts sometimes prepare a consensus estimate, This tends to minimize individual oversights and lack of familiarlity with particular objects a ,d neutralizes personal biases. The major disadvantage of group estimation is the effect that interpersonal group dynamics may have on individuals in the group. The Delphi technique can be used to overcome these disadvantages,

3.2.2 Delphi cost estimation

The Delphi technique can be adapted to software cost estimation in the following manner,

1. A co-ordinator provides each estimator with the System Definition document and a form for recording a cost estimate,
2. Estimators study the definition and complete their estimates anonymously. They may ask questions to the co-ordinator, but they do not discuss their estimates with one another.
3. The co-ordinator prepares and distributes a summary of the estimators responses, and include any unusual rationales noted by the estimators,
4. Estimators complete another estimate, again unonymousy, using the results from the previous estimate. Estimators whose estimates differ sharply from the group may be asked, anonymouly, to provide justification for their estimates.
5. The process is iterated for as many rounds as required, No group discussion is allowed during the entire process,

The following approach is a variation on the standard Delphi technique that increases communication while preserving anonymity,

1. The co-ordinator provides each estimator with a System Definition and an estimation form.
2. The estimators study the definition and the co-ordinator calls a group meeting so that estimators can discuss estimation issues with the co-ordinator and one-another,
3. Estimators complete their estimates anonymouly.
4. The co-ordinator prepares a summary of the estimates, but does not record/any rationales.

5. The co-ordinator calls a group meeting to focus on issues where the estimates vary widely.
6. Estimators complete another estimate, again anonymouly. The process is iterated for as many rounds as necessary.

After each round of estimation, the co-ordinator discussed with each estimator in order to resolve the differences. These discussions are necessary as it is required to arrive at a consensus estimate atleast after several rounds of estimates.

3.2.3. Work breakdown structures

Expert judgment and group consensus are top-down estimation techniques. The word breakdown structure is a bottom-up estimation tool. A work breakdown structure is a hierarchical chart that accounts for the individual parts of the system. A WBS chart can indicate either product hierarchy or process hierarchy.
In product hierarchy, product components are identified and the manner in which the components are iriter-connected are specified. A WBS chart of process hierarchy identifies the work activities and the relationships among those activities. Typical product and process WBS chart are illustrated in fig.3.4 a and b.

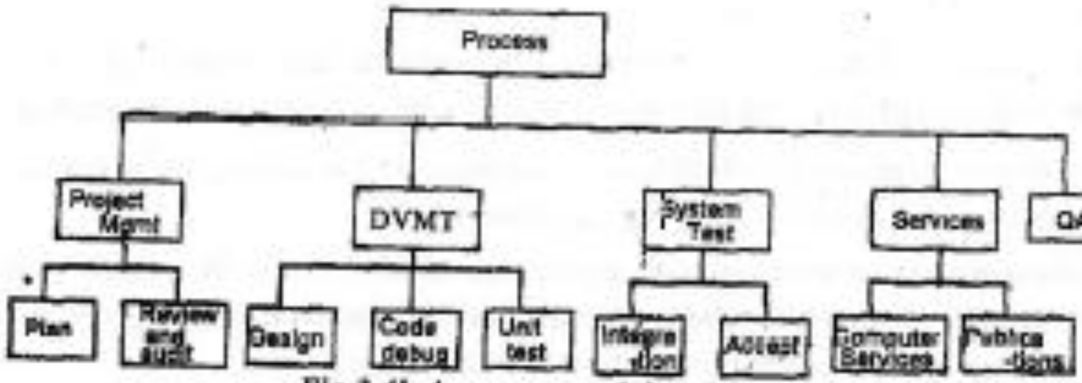


Fig 3.4b A process work breakdown structure

Using the WBS technique, cost are estimated by assigning costs to each individual component in the chart and summing the costs, The main advantage of using WBS cost estimation technique is that it makes all the process and product factors that are included in the cost estimation explicit to all.
Expert judgment, group consensus and WBS are most widely used cost estimation technique. Many organizations use all three approaches and iterate on the estimates until differences have been resolved.

3.2.4.Algorithmic cost models

Algorithmic cost models are bottom-up cost estimators. Algorithmic cost estimators compute the estimated cost of a software system as the sum of the costs of the modules and subsystems that comprise the system. The Constructive Cost Model (CoCoMo) is an algorithmic cost model described in BOE81 CoCoMo is briefly summarized here. The equations presented in section 3.1. are used when using CoCoMo to provide nominal estimates of programmer-months and development schedule for a program, unit, based on the estimated number of Delivered Source Instructions (DSI) in the unit Effort multipliers are then used to adjust the estimate fot product attributes, computer attributes personnel attributes and project attributes. Table 3.4 summarizes the cocomo effort multipliers and their range of values.

The cocomo incorporates a number of assumptions for example, the nominal application programs equations apply in the following types of situations.

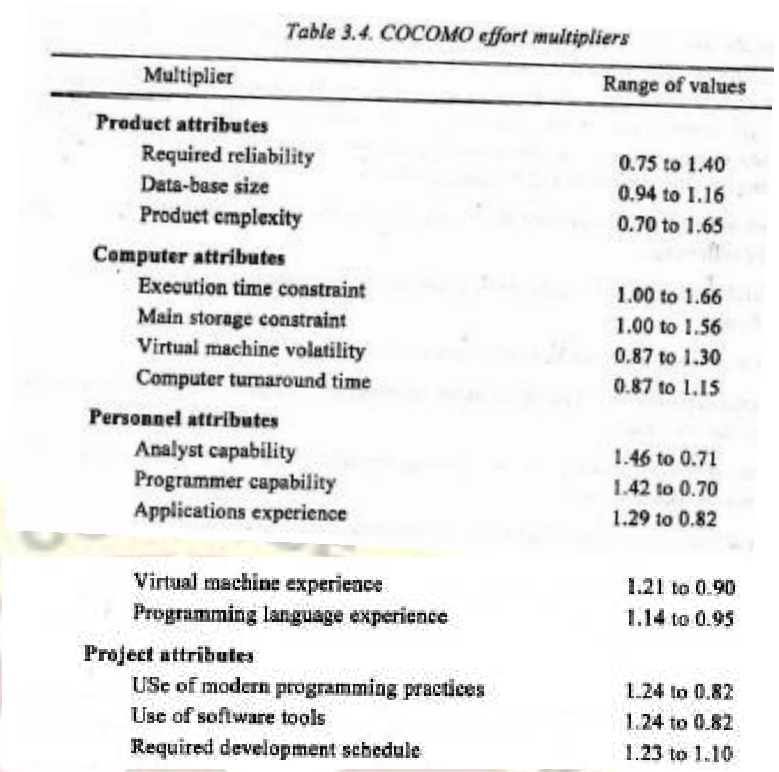


Table 3.4. COCOMO effort multipliers

Multiplier	Range of values
Product attributes	
Required reliability	0.75 to 1.40
Data-base size	0.94 to 1.16
Product cmplexity	0.70 to 1.65
Computer attributes	
Execution time constraint	1.00 to 1.66
Main storage constraint	1.00 to 1.56
Virtual machine volatility	0.87 to 1.30
Computer turnaround time	0.87 to 1.15
Personnel attributes	
Analyst capability	1.46 to 0.71
Programmer capability	1.42 to 0.70
Applications experience	1.29 to 0.82
Virtual machine experience	1.21 to 0.90
Programming language experience	1.14 to 0.95
Project attributes	
USe of modern programming practices	1.24 to 0.82
Use of software tools	1.24 to 0.82
Required development schedule	1.23 to 1.10

Small to medium size projects (2k to 32k DSI) Familiar application area Stable, well- understood virtual machine In-house development effort Effort multipliers are used to modify these assumptions The following activities are covered by the estimates. Covers design through acceptance testing

Includes cost of documentation and reviews includes cost of project manager and program librarian. The effort estimators exclude planning and analysis costs, installation and training costs, and the cost of secretaries and computer operators, The DSI estimate includes job control statements and source statements but excludes comments and unmodified utility routines. A DSI is considered to be one line or card image and a programmer- month consists of 152 Programmer-hours.

Other assumptions concerning the nature of software projects estimated by cocomo include the following.

Careful definition and validation of requirements is performed by a small number of capable people.

The requirements remain stable throughout the project.

Careful definition and validation of the architectural design is performed by a small number of capable people.

Detailed design, coding and unit testing are performed in parallel by groups of programmers working in teams,

Integration testing is based on early test planning,

Interface errors are mostly found by unit testing and by inspections and walkthroughs before integration testing documentation is performed incrementally as part of the development process.

From the above assumptions it is clear that systematic techniques of software engineering are used throughout the development process.

The following example of algorithmic cost estimation using cocomo is adapted from Boehm(BOE81).

The product to be developed is a 10-KDSI embedded mode software product (Systems programs) for telecommunications processing on a commercially available micro processors. The nominal estimates of programmer-months and development schedule are got from the equations presented in Section 3.1.

M = 2.8*(KDSI)** 1.20
= 2.8* (10)** 1.20
= 44.4

TDEV = 2.5 * (PM) * * 0.32 '
= 2.5 * (44) * * 0.32
= 8.4

From fig 3.1, it can be noted that the ratio of programmer-months is roughly 1 to 1.7 to 2.8 for development of application program, utility program and systems program respectively.

Effort multipliers are used to adjust the estimate for off-nominal aspects of the project. The effort multipliers for this project are presented in Table 3.5.
Table 3.5. Effort multipliers for embedded telecommunications example

Multiplier Rationale Value

Reliability Local use only.	
No serious recovery problems(nominal)	1.00
Data base 20,000 bytes (low)	0.94
Complexity Telecommunication processing (very high)	1.30,
Timing Will use 70% of processing (high)	1.11
Storage 45k of 64k available (high)	1.06
Machine Stable Commercially available micro processor (nominal)	1.00
Turnaround Two hours average(nominal) ..	1.00
Virtual machine experience	1.21 to 0.90
Programming language experience	1.14 to 0.95
Project attributes	
USe of modern programming practices	1.24 to 0.82
Use of software tools	1.24 to 0.82
Required development schedule	1.23 to 1.10

The cocomo incorporates a number of assumptions for example, the nominal application programs equations apply in the following types of situations.
Small to medium size projects (2k to 32k DSI)
Familiar application area
Stable, well- understood virtual machine
In-house development effort Effort multipliers are used to modify these assumptions
The following activities are covered by the estimates.
Covers design through acceptance testing

Includes cost of documentation and reviews includes cost of project manager and program librarian. The effort estimators exclude planning and analysis costs, installation and training costs, and the cost of secretaries and computer operators, The DSI estimate includes job control statements and source statements but excludes comments and unmodified utility routines. A DSI is considered to be one line or card image and a programmer- month consists of 152 Programmer-hours.

Other assumptions concerning the nature of software projects estimated by cocomo include the following.

Careful definition and validation of requirements is performed by a small number of •capable people.
The requirements remain stable throughout the project.

Careful definition and validation of the architectural design is performed by a small number of capable people.

Detailed design, coding and unit testing are performed in parallel by groups of programmers working in teams,

Integration testing is based on early test planning,

Interface errors are mostly found by unit testing and by inspections and walkthroughs before integration testing documentation is performed incrementally as part of the development process.

From the above assumptions it is clear that systematic techniques of software engineering are used throughout the development process.

The following example of algorithmic cost estimation using cocomo is adapted from Boehm(BOE81).

The product to be developed is a 10-KDSI embedded mode software product (Systems programs) for telecommunications processing on a commercially available micro processors. The nominal estimates of programmer-months and development schedule are got from the equations presented in Section 3.1.

M = 2.8* (KDSI)* * 1.20
2.8 * (10) * * 1 20
= 44.4 TDEV = 2.5 * (PM) * * 0.32 l
= 2.5 * (44) * * 0.32
8.4

From fig 3.1, it can be noted that the ratio of programmer-months is roughly 1 to 1.7 to 2.8 for development of application program, utility program and systems program respectively.

Effort multipliers are used to adjust the estimate for off-nominal aspects of the project. The effort multipliers for this project are presented in Table 3.5.

Local use only.	
No serious recovery problems(nominal)	1.00
20,000 bytes (low)	0.94
Telecommunication processing (very high)	1.30
Will use 70% of processing (high)	1.11
45k of 64k available (high)	1.06
Stable Commercially available micro processor (nominal)	1.00
Two hours average(nominal) .,.	1.00
Analysts Good senior people (high)	0.86
Programmers Good senior people (high)	0.86
Experience tl Three years in telecomm (nominal)	1.00
Experience Six months on the micro (loco)	1.10
Experience Twelve months with the language (nominal)	1.00
Practices More than 1 year experience with modern techniques (high)	
Tools Basic micro software (loco)	1.10
Schedule Nine months, 8.4 estimated (nominal)	1.00
Effort Adjustment factor =	1.17.

The effort adjustment factor of 1.7 is the product of the effort multipliers. When applied to the nominal estimate, the effort adjustment factor produces an estimate if 51.9 programmer-months and 8.8 months development time.

1. Identify all subsystems and modules in the product.
2. Estimate the size of each module and calculate the size of each subsystem and total system.
3. Specify module - level effort multipliers for each module. The module - level multipliers are product complexity, programmer capability, virtual machine experience and programming language experience.
4. Compute the module effort and development time estimates for each module using the nominal estimator equations **and** the module-level effort multipliers.
5. Specify the remaining 11 effort multipliers for each subsystem.
6. From steps 4 and 5, compute the estimated effort **and** development time for each subsystem
7. From step 6, compute the total system effort and development time.
8. Perform a sensitivity analysis on the estimate to establish tfade-off benefits.
9. Add other development costs, such as planning and analysis, that are not included in the estimate.
10. Compare the estimate with one developed by top-down Delphi estimation. Identify and rectify the differences in the estimates.

The main advantage of cocomo is that the cost factors within an organization is known. Data can be collected and analyzed, new factors can be identified and the effort multipliers can be adjusted as necessary. The greatest weakness of cocomo is that use of a multiplicative effort adjustment factor assumes that the various effort multipliers are independent. In reality, varying one factor often implies that other factors should also be adjusted. Often, it is not cleat how variations in one factor influence the other factors.

3.3. Staffing-Level Estimation

The number of personnel required throughout a software development project is ndt constant. The number of personnel required during each phase of the project differs from one project to the other. Typically, planning and analysis are performed by a small group of people, architectural design by a larger group and detailed design by a still larger number of people. The early phase of maintenance may require numerous personnel, but the number should decrease in a short time. If there is no major enhancement or adaptation, then the number of personnel for maintenance should remain small - Different authors have given many illustrations. We will discuss some of them in detail.

Norden observed that research and development projects follow a cycle of planning, design, prototype, development and use, with the corresponding personnel utilization illustrated

illustrated in fig.3.5.

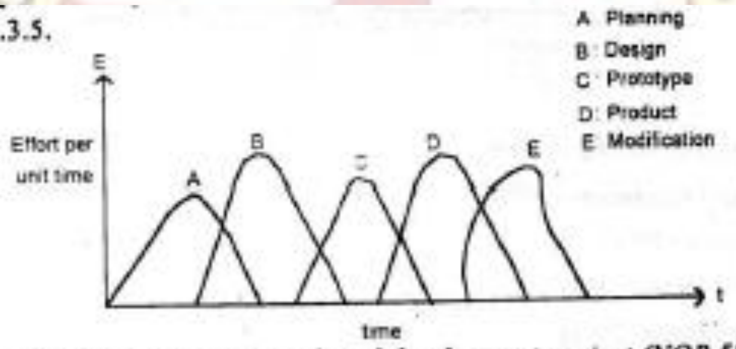


Fig. 3.5. Cycles in a research and development project (NOR 58)

His another observation is that the sum of the areas under the curves in fig 3.5.can be approximated by the Rayleigh equation as illustrated in fig 3.6. (Nor 58)

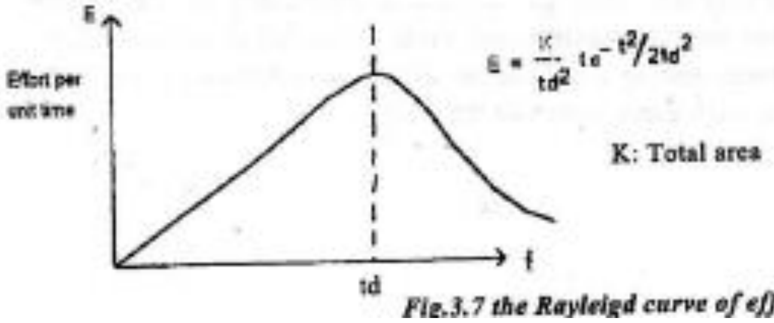
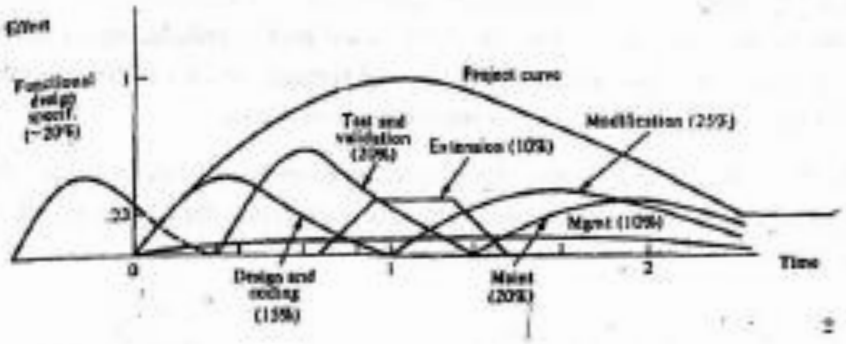


Fig.3.7 the Rayleigd curve of effort Vs time

The Rayleigh curve is specified by two parameters. td, the time at which the curve reaches its maximum value, and k, the total area under the curve, which represents the total effort required for the project. Any particular point on the Reyleigh curve represents the number of full-time equivalent personnel required at that instant of time.

According to Putnam, the time at which the Rayleigh curve reaches its maximum value,td. corresponds to the time system testing and product release for many software products. The area under the Rayleigh curve in any interval represents the total effort expended in that interval.

Approximately 40 percent of the area under the Rayleigh curve is to the left of td and 60 percent is to the right. For many product life cycles, this is a reasonable estimate of the distribution of effort between development and maintenance. Putnam's interpretation of the Rayleigh curve is illustrated in fig 3.7.



A: Design and coding (15%)

B: Test and validation (20%)

C: Extension (10%)

D: Modification (25%)

E: Maintenance (20%)

F: Management (10%)

Fig. 3.7 Putnam's interpretations of the Reyleigh Curve.

Boehm observes that the Rayleigh curve is a reasonably accurate estimator of personnel requirements for the development cycle from architectural design through implementation and system testing if the portion of the curve between 0.3 td and 1.7 td is used (BOE 81). The Rayleigh curve is then of the form.

$$FSP = PM\left(\frac{0.15\,TDEV + 0.7t}{0.25(TDEV)^2}\right)_e - \frac{(0.15TDEV + 0.7t)^2}{0.5(TDEV)^2}$$

Where

PM - estimated number of programmer months for product development (excluding planning and analysis)

TDEV - estimated development time

Given these two factors, the number of full-time software personnel, FSP, required at any particular time t, where t is in the range 0.3 td to 1.7td can be computed.

Boehm also presents tables that specify the distribution of effort and schedule in a software development project.The total number of programmer-months and total development time can be used to obtain an estimate of the actual number of programmer months and elapsed time for each activity. An estimate of the number of full-time development can be obtained by dividing the numbers of programmer - months required by the elapsed time available. Table 3.7 illustrates the distribution of effort, schedule and required personnel for a 32- KDSI, 91 PM, 14 months project and a 128 KDSI 292-PM, 24 month project.

Table 3.7 Distribution of effort, schedule and personnel

Activity	Effort		Schedule	32	Personnel	32
	32KDSI	128KDSI	KDSI	128KDSI	KDSI	128KDSI
Plans and requirements	5PM	24MM	1.2MO	3.1MO	2.9FSP	8FSP
Architectural design	15PM	63MM	2.20MO	4.6MO	5.6FSP	14FSP
Detailed design	22PM	90MM	Combined	Values		
Implementation	34PM	141MM	7.7MO	12.2MO	7.3.FSP	19FSP
System test	20PM	90MM	3.6MO	7.2MO	5.6FSP	14FSP

Estimating software maintenance costs.

Software maintenance typically requires 40 to 60 percent and in some cases as much as 90 percent of the total life - cycle effort devoted to software product. Maintenance activities include

1. Enhancements to the product.
2. Adapting the product to new processing environments.
3. Correcting problems.

During the planning phase of a software project thq number of maintenance programmers needed are estimated and facilities required for maintenance are specified. Lientz and Swanson determined that the typical level of effort devoted to software maintenance was around 50 percent of total life-cycle effort, and that the distribution of maintenance activities was 51.3 percent for enhancement. 23.6 percent for adaptation, 21.7 percent for repair and 3.4 percent for other (LIE 80) These percentages are further broken down in Table 3.8.

Table 3.8. Maintenance effort distribution (LIE 80)

Activity	%Effort
Enhancement	51.3
Improvement efficiency	4.0
Improved documentation	5.5
User enhancements	41.8
Adaptation	23.6
Input data, files	17.4
Hardware, operating system	06.2
Corrections	21.7
Emergency fixes	12.4
Other	3.4

The software requirements specification

The format of a requirement specification document is presented in table 3.9 *Table 3.9 Format of a software requirements specification*

Section 1: Product overview and summary
Section 2: Development, Operating and Maintenance Environments
Section 3: External Interfaces and data flow
Section 4: Functional Requirements
Section 5: Performance Requirements
Section 6: Exception Handling
Section 7: Early subsets and Implementation Priorities
Section 8: Foreseeable Modifications and Enhancements
Section 9: Acceptance criteria
Section 10: Design Hints and Guidelines
Section 11: Cross-Reference Index
Section 12: Glossary of terms

Sections 1 and 2 of the requirements document present an overview of product features and summarize the processing environments for development operation and maintenance of the product. This information is an elaboration in the software product characteristics contained in the system Definition and the preliminary users manual.

External Interfaces such as user displays and report formats, a summary of user commands and report options, data flow diagrams and data dictionary are included in section-3. High level data flow diagrams and a data dictionary are derived.

Data flow diagrams specify data sources and data sinks, data stores, transformations to be performed on the data, and the flow of data between sources, sinks, transformations and stores. A data store is a conceptual data structure in the sense that physical implementation details are suppressed; only the logical characteristics of data are emphasized in a data flow diagrams.

Data flow diagrams can be depicted informally, as illustrated in fig 3.8 or by using special notation as illustrated in fig 3.9. In fig 3.9 data sources and data sinks are depicted by shaded rectangles transformation by ordinary rectangles and

data stores by open ended rectangles. The arcs in a data flow diagram specify data flow. They are labeled with the names of data items. The characteristics of these data items are specified in the data dictionary.
Data flow diagrams are not concerned with decision structure or algorithmic details. Data flow diagrams can be used at any level of detail. They can be hierarchically diecomposed. The inner workings of the functional nodes are specified using additional data flow diagrams.

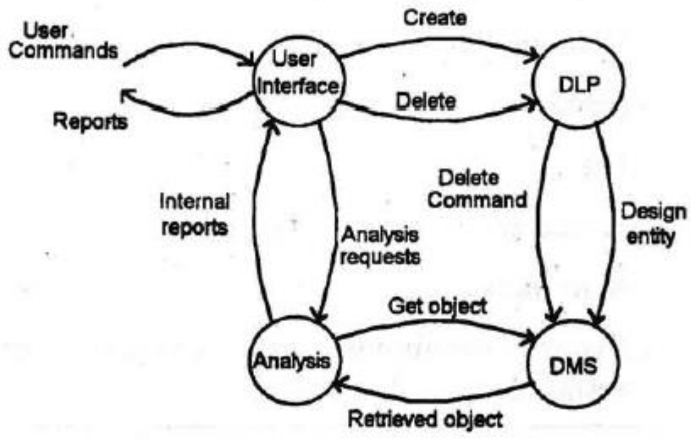


Fig. 3.8. An informal data flow diagram

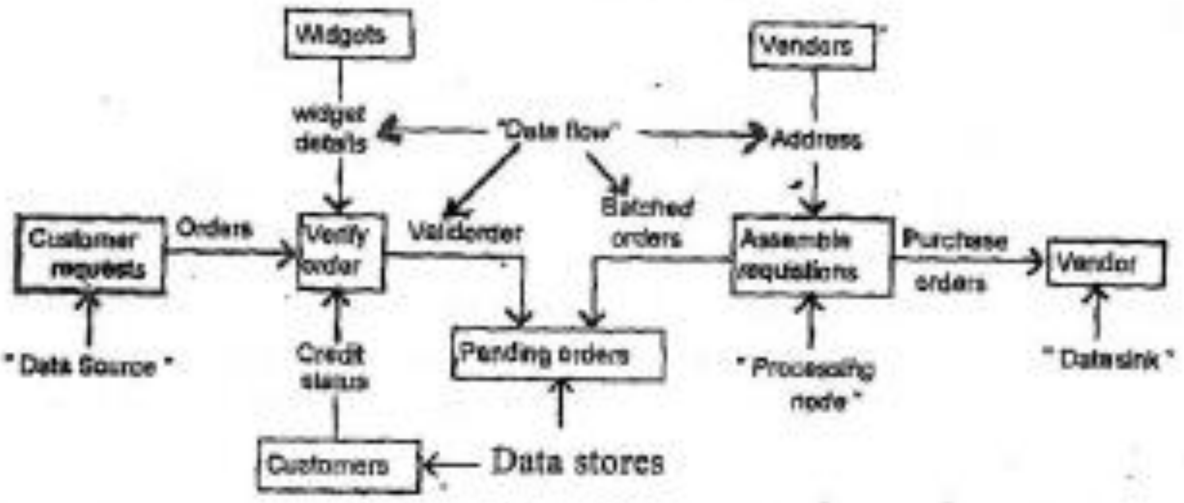


Fig. 3.9. A formal data flow diagram

A data dictionary is illustrated in Table 3.10. Each named data item on each data flow diagram should appear in the data dictionary. The physical implementation details are not of interest in the data dictionary.

Table 3.10 A Data dictionary entry

NAME:
WHERE USED:
PURPOSE:
DERIVED FROM:
SUB-ITEMS :
NOTES:
Create SDLP

Create passes a user-created design entity to the SLP processor for verification of syntax.
User Interface Processor
Name
Uses
Procedures
References
Create conditions one complete user-created design entity,

Functional requirements for the software product specified in section 4 of the software Requirements specification. Functional requirements atr typically expressed in relational and state oriented notations that specify relationships among inputs, actions and outputs.

Section 5 of the requirements document specifies the performance characteristics such as response time for various activities processing time for various processes, throughput primary and secondary memory constraints, required telecommunication bandwidth and special items such as extraordinary security constraints or unusual reliability requirements. Performance characteristics must be stated in verifiable items and the methods to be used in verifying performance must also be "specified. This must be stated in such a way that logical reasoning can be applied to the statement of requirements.

The actions to be taken and the messages to be displayed in response to uhdesired situations or events are specified in section 6 of the software Requirements specification. A table of exception conditions and exception responses should be prepared. Categories of possible exceptions include temporary resource failure (eg. temporary loss of a sensor or communication link); permanent resource failures (eg, loss of a memory bank or processor); incorrect, inconsistent, or out of range input data, internal values, and parameters, violation of capacity limits (eg. table overflow array indices out of range run time stack over-flow); and violations of restrictions on operators (eg. division by zero, attempt to read past end of file)

Section 7 of the software Requirements specification specifies early subsets and implementation priorities for the system under development. Software products are sometimes developed as a series of successive versions. Each successive version can *t* incorporate the capabilities of previous versions and provide additional processing functions. Sometimes the customer may desire to have early delivery of the software product with limited capability and may wish the successive versions to provide increasing levels of capabilities. The capabilities that must be included to each version must be planned initially. The subsets of the capabilities that are to be included to each successive version is specified in this section.

Depending on the importance of the system capabilities may be of an essential feature, desirable feature or it may be an"nice «f feature of the software product. The importance of the capabilities must be identified in order to provide guidance to the designers and implementors.

Section-8 of the requirements document specifies the foreseeable modifications and enhancements that may be incorporated into the product following the initial release. This specification of expected modifications and enhancements will enable the designers and implementors to design and build the product in a manner that will ease those changes. Foreseeable product modifications might occur as a result of anticipated changes in project budget, as a result of new hardware acquisition or as a result of experience gained from an initial release of. the product. Designing and implementing the product in view of the probable changes will ease the maintenance activity as the enhancements can be done very easily.

Section 9 of the requirements document specifies the software product acceptance criteria. Acceptance criteria specify functional and performance tests that must be performed, and the standards to be applied to source code, internal documentation and external documents such as the design specification, the test plan, the user's manual, the principles of operation, and the installation'and maintenance procedures. In addition' the desired functional and physical audits of source code, documents and physical media are specified. It is important that the acceptance criteria verify the functional and performance requirements stated in sections 4 and 5 of the product requirements.

Section 10 of the Software Requirements Specifications contains design hints and guidelines. During the planning phase, emphasis is placed on the product characteristics without implying how the product will provide those characteristics. Certain insights and Understandings may be gained during planning and requirements definition These are recorded as hints and guidelines to the designers, but not as rigid requirements for product design.

Some of the information contained in Requirements document is an elaboration of the software product characteristics contained in the System Definition and the preliminary User's Manual. In order to verify and re-examine the requirements,constraints, and assumptions, it is very much needed to know the sources of specific requirements.

Sectionl 1 relates product requirements. A cross-reference directory should be pnn ided to index specific paragraph numbers in the Software Requirements Specification to specific paragraphs in the System Definition and the preliminary User's Manual, and to other sources of information.

Section 12 of the. Software Requirements Specification provides definition of terms that may be unfamiliar to the customer and the product developers, Some standard terms may be used in non-standard ways. Care should be taken to include the definitions of these terms,

Desirable properties

There are a number of desirable properties that a Software Requirements Specification should process. In particular, a requirements document should be: correct complete consistent unambiguous Functional *H*
Verifiable TraceableEasily changed

If the requirements specification is incorrect and incomplete then the resultant software product will satisfy the requirements but not the customers need. An inconsistent specification states contradictory requirements in different parts of the document. An ambiguous requirement is subject to different interpretations by different people.

Software requirements should be functional in nature, i.e. what is required is specified without specifying the implementations details. This provides maximum flexibility for the product designers.

Requirements must be verifiable from two points of view, it must be possible to verify that the requirements satisfy the customer's need, and it must be possible to verify that the subsequent works products satisfy the requirements. Due to the lack of formal verification techniques for software requirements, the most important verification tool currently available is logical reasoning.

The requirements should be indexed, segmented and cross-referenced to permit easy use and easy modification. Every software requirement should be traceable to specific customer statements and to specific statements in the system definition.

Every project even after completion and the release expects modifications to be done to the product in order to meet the changing customer's need. Project success often depends on the ability to incorporate change without starting over.

Enhancements to the desirable properties of the Software Requirements Specification can be done using the formal notations and automated tools. These topics are discussed in the following sections.

Formal specification techniques

Formal notations are used to specify the functional characteristics of a software product in the requirements document. The requirements document will be concise and the functional characteristics can be expressed in an umambiguous manner by using the formal notations. These notations support formal reasoning about the functional specifications and they provide a basis for verification of the resulting software product. Formal notations are not appropriate in all situations or for all types of systems.

To specify the functional characteristics of software both relational and state oriented notations are used. Relational notations are based on the concepts of entities and attributes. Entities are named elements in a system, the names are.chosen to denote the nature of the elements (eg, stack queue). Attributes are specified by applying functions and relations to the names entitites. Attributes specify permitted operations on entities, relationships among entities and data flow between entities.

The state of system is the information required to summarize the status of system entities at any particular point in time. Given the current state and the current estimate, the next state can be determined. The execution history by which the current estimate, the next state, it is only dependent on the current state and current stimuli,

Relational notations include implicit equations, recurrence relations, algebraic axioms' and regular expressions. State oriented notations include decision tables, event tables, transition tables, finite-state mechanisms and Petri Nets. These notations are formal in the sense that they are concise and unambiguous. They are described in the following sections.

3.6.1.Relational Notations

Implicit equations: Implicit equations specify the properties of a solution without stating a solution method. Matrix inversion is specified as follows

$Mx_M' = I + E$ (31)

Matrix inversion has the property that the original matrix (M) multiplied by its inverse (M^1) yield an identify matrix. This property of the matrix inverse is specified in equation (3.1) I denotes the identify matrix and E specifies allowable computational errors Complete specification of matrix inversion must include items such as matrix size, type of data elements and degree of sparseness (ie) how many elements in the matrix are zeroes Given a complete functional specification for matrix inversion, design involves specifying a data structure and an algorithm for computing the inverse.

Implicit specification of a square root function, SQRT, can be stated as

$(0<=_X<=y)$ [ABS(SQRT(X)**2-X)<E] (3.2)

Equation (3.2) states that the square root of X after squaring it and substacting the value of X result in an error value that is in some permissible error range. The value of X is in the closed range 0 to Y.

Not all implicitly specified problems are guaranteed to have algorithmic solution For example a variant of Fermat's last problems can be stated as follows.

(N>2) [X**N+Y**N=Z**N] (3.3)

Fermants problem involves finding values of integers X,Y,and Z such that for arbitrarily chosen values of N, Equation (3.3) is satisfied. No solutions to this problem have been discovered for values of N greater than 2.

Recurrence relations

A recurrence relation consists of an initial part called the basic and one or more recursive parts. The recursive parts describe the desired value of a function in terms of other values of the function for example, successive fibonacci numbers are formed as the sum of the previous two fibonacci numbers. The initial part specifies the frrst two terms in the fibonacci series and they are 0 and 1.

F(0) = 0
F(l)=l
F(N) = F(N-1) + F(N-2) for ail N>1

Recurrence relations are easily transformed into recursive programs."That doesn't mean every recursive specification should;d be implemented as a recursive programs.

Algebraic Axioms

Mathematical systems are defined by axioms. The axioms specify fundamental properties of a system and provide a basis for deriving additional system and provide a basis for deriving additional properties that are implied by the axioms. These additional properties are called theorems. The set of axioms must be complete and consistent, i.e. it must not be possible to prove contradictory results.

The axiomatic approach can be used to specify functional properties of software systems. The intention is to specify the fundamental nature of the system by stating a few basic properties. This approach can be used to specify abstract data types. A data type is characterized as a set of objects and a set of permissible operations on those objects. The term "abstract data type" refers to the fact that permissible operation on the data objects are emphasized while representation details of the data objects are suppressed.

Abstraction allows one to emphasize the important characteristics of a system while suppressing unimportant details. Thus, data abstraction emphasizes functional properties and suppresses representation details. Depending on the axioms provided, only some of many possible functional properties are specified.
Specification of an abstract data type using algebraic axioms involves defining the syntax of the operations and specifying axiomatic relationships among the operations. The syntactic definition specifies names, domains and ranges of operation to be performed on the data objects and the axioms specify interactions among operations.

As an example axiomatic specification of the last-in-first-out (LIFO) property of stack objects is specified in fig.3.10

SYNTAX
OPERATION
NEW
PUSH
POP
TOP
EMPTY

DOMAIN RANGE
( ) -> STACK
(STACK.ITEM) -> STACK
(STACK) -» STACK
(STACK) -> ITEM

(STACK) -> BOOLEAN

AXIOMS

(Stk is of type STACK, itm is of type ITEM)

l.EMPTY(NEW) = true 2.
EMPTY(PUSH(Stk,itm)) = false .
_3.POP(NEW) = error
4. TOP(NEW) = error
5. POP(PUSH(stk,itm)) = stk
6. TOP(PUSH(stk,itm)) = item

Fig 3.10.Algebraic specification of the LIFO property.

In the algebraic specification of the LIFO property the type of data elements being manipulated, ITEM is a parameter in the specification and that the specification is representation independent ie the type of values these data elements take is not specified. Definitions of the stack operations are:

NEW creates a new stack
PUSH adds a new item to the top of a stack
TOP returns a copy of the top item
POP removes the top item q
EMPTY tests for an empty stack.

Operation NEW creates a newly created empty stack, PUSH operation requires two parameters namely STACK and 'TEM, which yield a STACK as a result with the item being included as the topmost element in the stack. POP operation has STACK as its parameter and it returns the stack as the result after removing the topmost element. Top operation returns the topmost item of the stack where the stack is the one and only parameter of the stack. EMPTY operation takes the stack as the parameter and returns a boolean value specifying whether the stack as the parameter and returns a boolean value specifying whether the stack is empty or not.

In the axiomatic definition of a stack presented in Fig. 3.10.it is assumed that stacks do not overflow; thus PUSH operation always succeeds. The nested entities provide the proper types as arguments for the entities in which they are nested. For example, in the axiom POP(PUSH(stk,itm) = stk PUSH yields type STACK, which is the required argument type for POP.

The axioms in fig 4.3.can be stated an English as follows:
1. A new stack is empty.
2. A stack is not empty immediately after pushing an item onto it,
3. Attempting to pop a new stack results in an errors.
4. There is no top item on a new stack.
5. , Pushing an item onto a stack unchanged.
6. Pushing an item onto a stack and immediately requesting the top item returns the items just pushed onto the stack.

Thus, we can see that the fundamental characteristics of stacks can be stated in English instead of using the axiomatic specification. The axiomatic specification has the advantage that the characteristics can be specified in a precise and unambiguous manner. Further more, the axioms can be manipulated in a rigorous manner.

The intuitively defined entities NEW, PUSH, POP, **TOP and** EMPTY are precisely defined in fig 4.4. using a state-oriented approach. In fig 4.4.an entity delimited by quotes [eg. 'valid - Stack(stk')] denotes the state of the system immediately before the operation in which it is contained. As unquoted entity refers to the state of the system immediately following the containg operation an exception condition has been specified for stack overflow in the definition of the PUSH function.

NEW

Purpose : Create a new stack.
Exceptions: Memory- full = true.
Effects: Valid-Stack(NEW) = 0
EMPTY(stk) *K*
Purpose : Test stk for empty property.
Exceptions: 'VAlid-Stack(stk)' = false
Effects: if 'Number-Items(stk)' = 0 then true else false
PUSH (stk,item)

Purpose:
Exceptions: **t**
Effects:
POP(stk) Purpose: Exceptions:
Effects: :
Place item on stk
'valid-Stack(stk)' = false
Number-Items(stk)' = MAX
if'Number-Items(stk)' = MAX then error
else Number-Items(stk) = 'Number-Items(stk)'+1
Fig 3.11. Definition of STACK function behavior
Delete top item from stk 'valid-stack(stk) = false 'Number-items(stk) = 0 if Number-ltems(stk)'=0 then error else {stk = 'stk'-TOP(stk)Number-Items(stk) = Number-item(stk)'-l}
TOP(stk)
Purpose:
Exceptions:
Effect:

Return a copy of a top item on stk. 'Vaild - Stack(stk)' = false 'Number-Items(stk)' = 0 if Number-Items(stk)' then error else Number-Items(stk)' = 'Number - Items(stk)'} *Fig. 3.11. Definition of STACK function behaviour (continuation)*

Using a state-oriented notation in conjunction with algebraic axioms allow precise specification of the entity names used in the axioms. This technique combines the advantages of the algebraic approach (precise specification of interactions among operations and the finite-state approach precise specification of the behavior of the individuals operations) From fig 3.10 it is eery clear that algebraic specifications emphasizes the interaction among the operations and frQm fig 3.11 it can be noted that state-oriented notations precisely specifies the behavior of the individual operations.

The definitions in fig 3.11.use primitive functions Valid-Stack and Number-Hems. These functions perform the self-evident actions denoted by their names. Formal definitions for these functions can be given, but at some point the fundamental definitions n all formal systems must involve symbols whose meanings are untrutively understood.

Given a set of algebraic axioms, new operations can be defined in terms of existing ones. For example, REPLACE can be defined to have the property of replacing the top data item on a stack with a new item.
REPLACE (stk,item) = if EMPTY(stk) then error else PUSH(POP(stk),item)
The theory of algebraic specifications is stated in terms of "constructors" ,"modifiers" and "behaviors". In fig 3.10 the constructors are NEW and PUSH, POP is a modifier and TOP and EMPTY are behavior. In order to provide a "sufficiently complete" set of axioms it is necessary to provide an axiom of the form. Modifier(Constructor()) = ? and Behavior(Constructor()) = ? for data abstraction

techniques are not limited to specification of simple abstractions such as stack and queues but can be used to define complex hierarchies of entities.

Algebraic axioms can be used in three distinct ways as definitional tools; as foundations for deductive proofs of desired properties and as frame works for examining the completeness and consistency of functions requirements. Illustrations has been given how algebraic specification can be used as definitional tools.

As an example of deductive reasoning about abstract data types, consider the definition of the stack operator REPLACE.
REPLACE(stk,itm) = if EMPTY(stk) then error else PUSH(POP(stk),itm)
Assuming that stk is not empty, the definitions provided in fig 4.4. can be used to rigourously demonstrate that REPLACE has the effect of replacing the top item on a stack with a new item \; ie. REPLACE (stk * itml.itm2) = stk * itm2.

Where stk * denotes all of the stack accept the top item and stk * itm denotes the entire stack.
REPLACE (stk *.itml,itm2) = PUSH(pop(stk*.iteml),itm2
= PUSH(stk*.itml-TOP(stk*.itml),itm2) .$_{tJ}$. = PUSH(stk *.itml-itml,itm2)
= PUSH(stk*,itm2)
= STK*.itm2 There are two types of completeness concerns for a requirement specification. External completeness Internal completeness

A requirement specifications is externally complete if all the desired properties are specified In fig 3.10 there is no specification for a SIZE function to return the current number of elements in the stack. If the.application requires SIZE, the specification is incomplete. External completeness is concerned with the question of whether the customer's needs are satisfied by a software product that incorporates all the stated specifications.

A requirements specification is internally complete if all the entities **ift** the specification has the definition. For example, use of the terms TOP without defining its meaning would result in a specifications for STACK that is internally incomplete. „

Consistency involves the relation ships among specifications. If, for example, the TOP operation for stacks is sometimes assumed to delete the top element of a stack and sometimes assumed to retain the top element, then TOP is used inconsistently.

Symbolic execution techniques can be used to "execute" algebraic specifications and report the result obtaned, or report error conditions encountered in attempting a sequence of operations.
For example, applying the sequence
1. S:=NEW 1. S is an empty stack
2.PUSH(S,10) 2. **s**
3.POP(PUSH(S,20)) 3. a. S b. S
After performing After performing the PUSH operation the POP operation
4.TOP(S) 4. returns 10 as the result as it is the topmost item in the stack
The following sequence
l.S:=NEW
2.PUSH(S,10)
3. POP(S)
4. POP(S) will result in an underflow error and an empty stack.

Symbol execution can be used to investigate completeness and consistency of algebraic specifications. The software engineer can exercise the specifications using a symbolic execution tools to understand how the specifications underact and to discover incompleteness and insconsistencies.

If an axiomatic specification is complete and consistent, any implementation that satisfies the stated relationships among operations will embody the desired characteristics. A complete and consitent set of functional specifications provides a complete and consistent set of functional characteristics of a software system are completely and consistently specified and if the implementation satisfies those specifications, there is no need for further testing or verification of the source code, except to verify performance, and to cerify that the code does not produce additional, undesired side effects.

Regular expressions

Regular expressions can be used to specify the syntactic structure of symbol strings. As many software products involve processing of symbol strings, regular expressions provide a powerful and videly used notation in software engineering. Every set of symbol strings specified by a regular expressions defines a formal language. Regular expressions can thus be viewed as language generators.

The rules for forming regular expressions are as follows.

1. Axioms: The basis sysmols in the alphabet of interest form regular expressions.
2. Alternation: If Rl and R2 are regular expressions, then (R1/R2) is a regular expression.
3. Composition: If Rl and R2 are regular expressions, then (R1.R2) is a regular expressions.
4. Closure: If Rl is a regular expressions, then (Rl)* is a regular expression.
5. Completeness: Nothing else is a regular expression.

Rule 1 is the basic rule. It states that an alphabet of basic symbols provides the atoms. The alphabet is made up to whatever symbols are of interest in the particular application.

Rules 2,3 and 4 are recursive. Alternation.(R1/R2). denotes the union of the language formed by concatenating strings from R2 onto strings from Rl. Closure,(Rl)*, denotes the language formed by concatenating zero or more strings from Rl with zero or more strings from Rl. Rule 5 complete the definition of regular expressions.

Examples of regular expressions follow:

1. Given atoms a and b, then (a/b) denotes the set {a,b}
2. Given atoms a,b and c, then ((a/b)/c) denotes set {{a,b},c}
3. Given atoms a and b and c , then ((a/b/c) (a\b) denotes the set {{a,b},c}.{a.b] containing one element ab.
4. Given atoms a and b then (a,b) denotes the set {ab} containing one element ab:
5. Given atoms a,b and c, then ((a,b),c) denotes the set {abc} containing one element abc.
6. Given atom a, the (a)* denotes the set {e,a,aa,aaa, }, where 1 denotes the empty string.

Complex regular expressions can be formed by repeated application of recursive rules 2,3 and 4.

1. (a(b/c)) denotes {ab,ac}
2. (a/b)* denotes {e,a,b,aa,ab,ba,aab,....}
3. ((a(b/c)))* denotes {e,ab,ac,abac,acab,abab,acac,ababac,....}

Closure, (Rl)*, denotes zero or more concatenations of elements from Rl. The notation (Rl)+ denotes one or more concatenations of elements in Rl. The "*" and "+" notations are called the kleene star and kleene plus notations.

Regular expressions can be given many different intrepretations and are thus useful in many different situations. For instance, ((a,(b/c)))+_ might denote any of the following.

1. A data stream. If a,b, and c are input data symbols, then valid data streams must always start with an "a", followed by "b" s and c"s in any order, but always unterleaved by "a" and terminated by "b" or V

2. Operation sequence. If a,b and c represent procedures then legal calling sequences are "a" followed by a call to "b" or "c" followed by zero or more reterns to "a" followed by calls to "b" or "c".

Hierarchical specifications can be constructed by assigning names to regular expressions and using the names in other regular expressions.

Regular expressions notaton can be extended to allow modeling of concurrency. By definition the effect of concurrent execution of two software components PI and P2 is the same if the execution histories of the two software components pi and P2 are interleaved. Interleaving of the regular expression for PI and P2 can be specified using an operation known as the "shuffle operator".

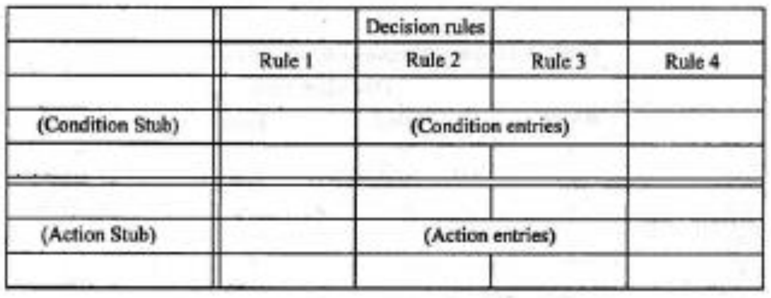
Path expression are another useful notation based on regular expressions. They can be used to specify the sequencing of operation in concurrent systems.
Another important application of regular expressions is in defining user interface dialogues. For example a user command string might be of the form command name. Name, followed by one or more white spaces, ws, followed by zero or more options, followed by a param-List.
Command-String = Name ws + (options ws+) * param-list
options = '-' option (ws+"-" option)*
Param -List = name(ws+Name)*
Name = Letter (Letter/Digit) *
WS = "

In the specification of command string symbols contained in single quotes are literals that can appear in the command strings. A complex specification for command string would include specifications for the syntax of option, Letter and Digit.

3.6.2. State Oriented Notations

Decision tables provide a mechanism for recording complex decision logic. Decision tables are widely used in data processing application. A decision table is segmented into four quadrants, condition stub condition entry, action states action entry as illustrate the condition state contains all of the conditions being examined. Condition entries are used to continue conditions into decision rules. The action state describe the action to be taken in responce to decision rules and the action entry quadrant relates decision rules to actions.

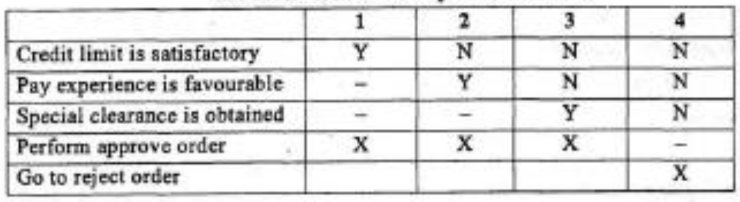
Table 3.11. Basic elements of a decision table:



		Decision rules		
	Rule 1	Rule 2	Rule 3	Rule 4
(Condition Stub)		(Condition entries)		
(Action Stub)		(Action entries)		

The format of a limited entry decision table is illustrated in table 3.12 The entries in it are limited to y, N and X. In a limited entry decision table, V denotes "Yes", N denotes "No".

Table 3.12Limited entry decision table

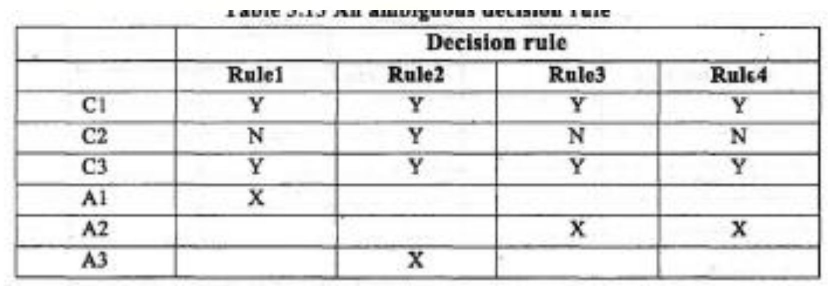


	1	2	3	4
Credit limit is satisfactory	Y	N	N	N
Pay experience is favourable	–	Y	N	N
Special clearance is obtained	–	–	Y	N
Perform approve order	X	X	X	–
Go to reject order				X

denotes"don't care" and X denotes "perform action . According to table 4.4.orders are approved if the credit limit is not exceeded, or if the credit limit is exceeded but past experience is good, or if a special arrangements has been made, if none of these conditions hold the order is rejected.

The entries in each column of the condition entry quadrant form a decision rule. Based on the entries in the column the required action to be performed is specifed action entry quadrant. The entries in the column specifies the conditions that are to be satisfied in order to perform the corresponding action. That's why the entries in each column is treated as a decision rule. If more that one decision rule has identical (Y,N) entries the table is said to be ambiguous. Ambiguous pairs of decision rules that specify identical actions are said to be redundant and those specifying different actions are contradictory. Contradictory rules permit specification of nondeterministic and concurrent actions. Table 3.13 illustrates redundant rules (R3 and R4) and contradictory rules **(Rl** and R3 and Rl and R4)

Table 3.13 An ambiguous decision rule



	Decision rule			
	Rule1	Rule2	Rule3	Rule4
C1	Y	Y	Y	Y
C2	N	Y	N	N
C3	Y	Y	Y	Y
A1	X			
A2			X	X
A3		X		

If every possible set of conditions has a corresponding action prescribed then the decision table is said to be complete. There are 2** N combinations of conditions in a table that has N condition entries. Failure to specify an action for any one of the combinations results in an incomplete decision table. Failure to specify an action for any one of the combinations results in an incomplete decision table. For example, in Table 3.14.the combinations (N,N,N) for conditions cl, c2 and c3 has no action specified. The condition (Y,Y,N) specifies both actions Al and A2. This multiply- specified action may be desired or it may indicate a specifications error.

Table 3.14 An incomplete and over-specified decision table

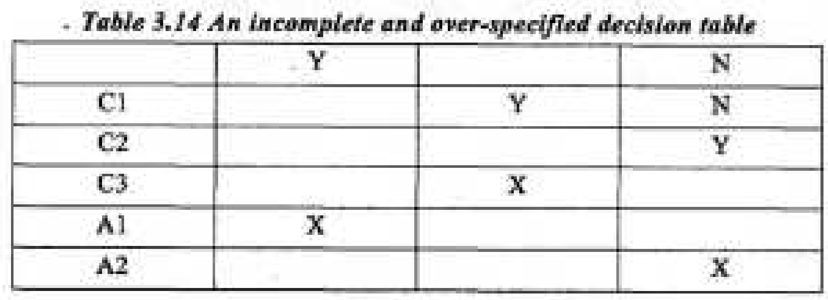
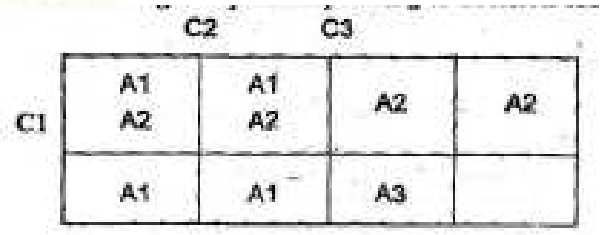


Table 3.14 An incomplete and over-specified decision table

	Y		N
C1		Y	N
C2			Y
C3		X	
A1	X		
A2			X

Fig 3.12 illustrates the use of a Karnaugh map its check a decision table for completeness and multiply specified actions.

Fig 3.12 Karnaugh map corresponding to decision table 3.14

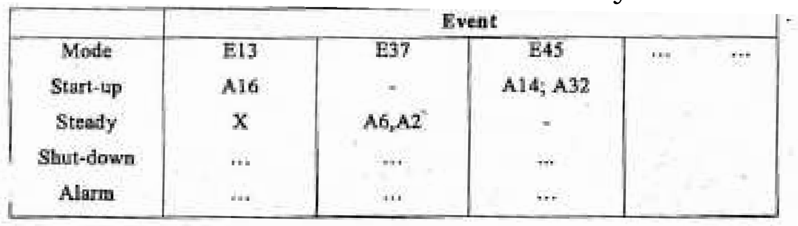


	C2	C3		
C1	A1 A2	A1 A2	A2	A2
	A1	A1	A3	

The specification is incomplete if there are any blank entries in the Karnaugh map. The specification is multiply-specified if there are any multiple entries in the Karnaugh map.

Event Tables: Event tables specify actions to be taken when events occur uner different set of conditions. Event tables are viewed as two- dimensional tables or of higher dimension. The entries in the table specify the action to be taken in the case of an event occuring under a set of conditions. Actions are related to two variables. f(M,E) = A, where M denotes the current set of operating conditions. E is the event of interest and A is the action to be taken.

Table 3.15 illustrates an event table where the actions to be taken are related to the current mode of operation and the events that may occur within modes. Thus, if the system is in start-up mode SU and event El3 occurs, actions A16 is to be taken, if(SU,E13) = A16.Special notations can be invented to suit particular situations. For example, actions seperated by semicolons (A14,A32) might denote A14 followed sequentially by A32. while actions seperated by commas (A6,A2) might denote concurrent acuration of A6 and A12. Similarly, a dash(-) might indicate no action required, while an X might indicate an impossible system configuration (eg. El3 cannot occur in steady-state mode).



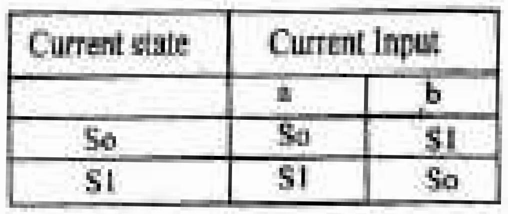
	Event				
Mode	E13	E37	E45	...	...
Start-up	A16	-	A14; A32		
Steady	X	A6,A2	-		
Shut-down	...	...	...		
Alarm	...	...	...		

Table 3.15 A two- dimensional event table:

Tranisition tables:

Transition tables are used to specify changes in the state of a system as a function of during forces. The state of a system summarizes the status of all entitles in the system at a particular time. Given the current state and the current conditions, the next state results: .t when in state Si condition Cj results in a transition to state Sk, we say f(Si,Cj) - Sk

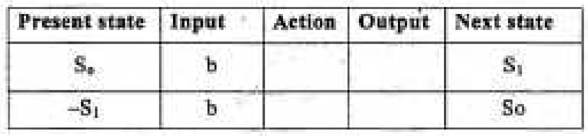
The format of a simple transistion table is illustrated in Table 3.16.



Current state	Current Input	
	a	b
So	So	SI
SI	SI	So

Table 3.16 A Simple Transition Table

In table 3 16 given current state So and current input b, the system will go to state SI. That is represented as f(So,b) = SI. Table 3.17 illustrates ^transition table that is augmented to indicate actions to be performed and outputs to be generated in the transition to the next state.



Present state	Input	Action	Output	Next state
S_o	b			S_1
$-S_1$	b			So

Table 3.17 An augmented transition table

An alternative representation for transition tables are transition diagrams. Transition diagrams are represented using directed graphs where the nodes in the graph represents the states of the system and the arcs connecting the nodes represent the transition of the state, from one state to another. Arcs are labeled with conditions that cause transitions. Fig 3.13.illustrates the transition diagram corresponding to the transition table in Table 3.17.

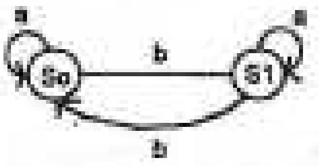


Fig. 3.13. Transition diagram corresponding to Table 4.8

Decision tables, event tables and transition tables are notations for specifying actions as functions of the conditions that imitiate those actions. Decision tables specify actions in terms of complex decision logic event lables relate actions to system conditions and transition tables incorporate the concept of system state. A specification in one of the notations can be expressed in the other two, The best choice among tabular forms for requirements specification depends on the particular situation being specified.

Finite State mechanisms: Data flow diagrams, regular expressions, and transition tables can be combined to provide a powerful finite state mechanism for functional specification of software system.

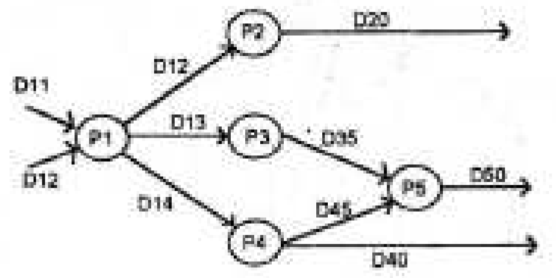


Fig 3.14 A network of data streams and processes

Finite state mechanisms utilize data flow diagrams in which the data streams are specified using regular expressions and the actions in the processing nodes are specified using transition lables. Fig 3.14.depicts the data flow diagram for a software system consisting of a set of processes interconnected by data streams. Each of the data streams can be specified using a regular expression and each of the processes can be described using a transition table.

Not only simple data streams can be expressed using regular expressions but also complex data streams. Fig 4.8.specifies a system for which the incoming data stream consists of a start marker DS followed by zero or more Dll messages, followed by zero or more D12 messages, followed by an end-of-data marker De. The regular expression for the incoming data stream isDsDll *D12* DE

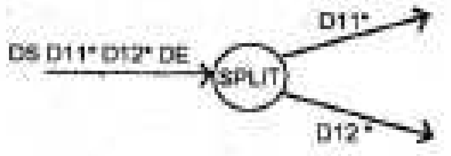
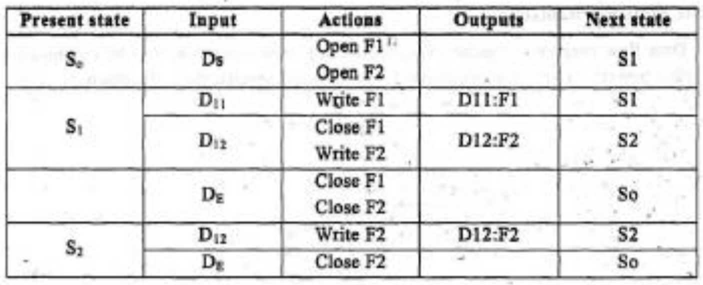


Fig. 3.15. Data flow diagram of SPLIT process.

The purpose of the process SPLIT is to route Dll messages to file Fl and D12 messages to file F2. The transition table in fig. 3.16 specifies the action of "split". Process split starts in initial state, So and wait for input Ds. Any other input in state So is ignored. Arrival of input Ds in state So triggers the opening of files Fl and F2 and transition to state SI. In SI, Dll messages are written to Fl until either a D12 message or a De message is received. On receipt of a D12 message, process split loses Fl, writes zero or more D12 messages to F2 and returns to state So to await the next transmission.



Present state	Input	Actions	Outputs	Next state
S_o	Ds	Open F1 Open F2		S1
S_1	D_{11}	Write F1	D11:F1	S1
	D_{12}	Close F1 Write F2	D12:F2	S2
	D_E	Close F1 Close F2		So
S_2	D_{12}	Write F2	D12:F2	S2
	D_E	Close F2	-	So

Fig 3.16. Transition table specifying- the action of "SPLIT"

The main drawback of finite-state mechanisms is the so-called state explosion phenomenon. Complex systems may have large numbers of states and many conditions of input data. Specifying the behaviour of a software system for all combinations of current state and current input will be tedious.

Hierarchical decomposition is one techique for controlling the complexity of a finite-state specification.

Using hierarchical decomposition higher level concepts are given names, the meaning and validity of which are established at a lower level.

Petri nets:

Petri nets have been used to model a wide verify situations: they provide a graphic?! representation technique and systematic methods have been developed for synthesizing and analyzing petri nets. Petri nets were invented to overcome the limitations of finite state mechanisms in specifying parallelism.

Petrinets are widely used to model various aspects of concurrent systems. Concurrent systems are designed to permit simultaneous execution of the software components, called tasks or processes, on multiple processors. Alternatively,

execution of tasks can be interleaved on a single processor. The fundamental problem of concurency are
a. Synchronization
b. mutual exclusion

Concurrent tasks must be Synchronized to permit communication among tasks that operate at differing execution rates, to prevent simultaneous updating of shared data and to prevent deadlock. Deadlock occurs when all tasks in the system are waiting for data or other resourses that can only be supplied by tasks that are also waiting on other tasks.

A petri net is represented as a bipartite directed graph. There are two types of nodes in a Petri net called places and transitions.
Places are marked by tokens. A petri net is characterized by an intitial marking of places and a firing rule. A firing rule has two aspects; a transition is enabled if every input place has atleast one token. An enabled transition can fire, when a transition firest, each input place of that transition loses one token and each output place of that transition gains one token. A marked petri net ia formally defined as a quaduaple consisting of a set of places P, a set of transitions T, a set of arcs A and a marking M.
C = (P,T,A,M)
Where P = {pl,p2, pm}
T= {tl,t2, to}
A C {P x T} U {T x P} = {(pi,tj) (t_r,P_L) }
M : P '-* I; ie M (pl,p2 pm) = (il,i2 im)
Marking M associates an integer number of tokens ik with each place pk.
When a transition fire, the marking of places p changes from M(p) to M'(p) as follows:
M^P) = M(P) +1 if P **e** O(t) and p g I(t)
$M,^1(P)$ = M(P)-1 ifp g 0(t)andp **e** I(t)
M^P) = M(P) otherwise.
Where I(t) is the set of input places of transition t and 0(t) is the set of output places of transition t. A transition is enabled if M(p)>0 for all Pe 1(f)
Fig 3.17 specifies the petri net model of concurrent processes. The three concurrent processes are marked as transitions tl,t2 and t3. The initial marking is specified in fig 3.17. In that place Po alone is having only one token and all the other places are not having any token. The functioning of the system is as follows. As the place Po is having one token and as it is the only one input place for the transition to, it is enabled. The enabled transition can

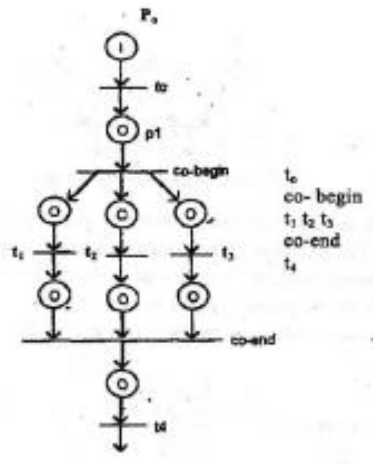


Fig 3.17 Petri net model of concurrent processes t1,t2,t3 (initial marking)

Fig 3.18 illustrates a deadlock situation

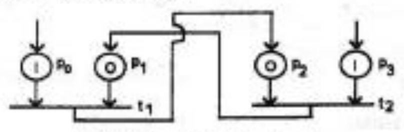


Fig 3.18 a deadlocked petrinet

fire and the firing removes the token in PO and places a token in PI. This enables the transition co-begin which removes the token in PI and places a token on each of P2, P3. This enables the transitions tl,t2 and t3; they can fire simultaneously or in any order. This corresponds to concurrent execution of tasks tl,t2 and t2UWhen each of tasks tl, t2 and t3 completes its processing, a token is placed on the corresponding output place, P5,P6 and P7. Co-end is not enabled until all three tasks from P5,P6 and P7 and places a token on P8 thus enabling t4 which can then fire.

The input places for the transition tl are PO and PI, the input places for the transition t2 are p2 and p3. In fig 4.11 places pO and p3 are having one token each. For the transition tl to get enabled, the place pi must have a token. But the place pi is the output place for the transition t2. It will get a token only on the firing of t2. The transition t2 will fire only if p2 is having a token p2 is an output place for the transition tl.p2 will receive a token only if the transition tl fires. So the transition tl and t2 are deadlocked as they are waiting for the others to fire and neither can proceed.

Conflict in Petri nets provides the basis for modeling of mutual exclusion. Conflict is illustrated* in fig 3.19, both tl and t2 are enabled, but only one can fire. Firing one will disable the other.

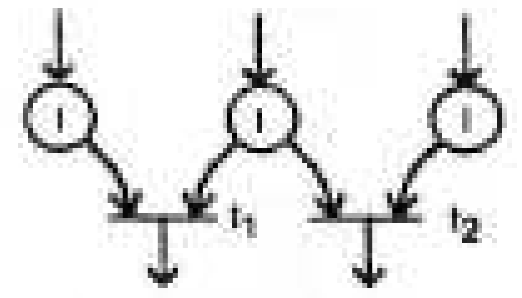
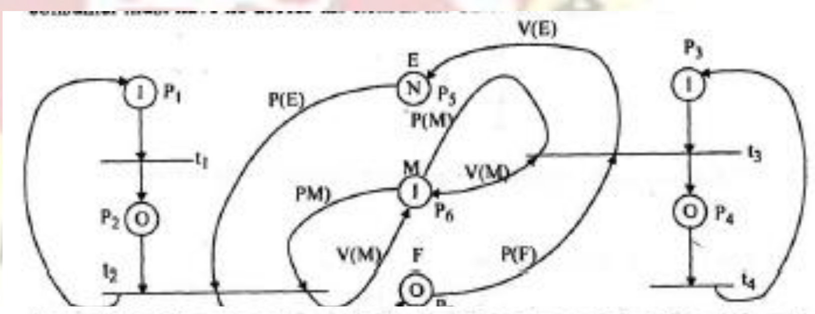


Fig 3.19 Conflicts in Petric nets

As an example of mutual exclusion, a Petrinet of the producer/consumer problem is illustrated in fig 3.20. The problem is the producer will be producing the items and places in the buffer. The consumer will take the item from the buffer and consumes it. The buffer of items is shared both by the producers and by the consumer. Both the producers and the consumer must have no access the item in the buffer at the same time.

tl : Produce V(F)
t2: Place in buffer

t3: Remove from buffer
t4: Consume



Transition	Input place(s)	Output Place(s)
t1	p1	p2
t2	p2, p5, p6	p1, p6, p7
t3	p3,p6,p7	p4,p5,p6
t4	p4	p3

Fig 3.2Q Initial marking for the producer/consumer PetriNet

Here tl is the producing process. t2 places a produced item in the buffer. t3 removes an item from the buffer, and t4 consumes it. The number of tokens on place E indicates the current number of empty buffe* positions. Initially E has a member of tokens equal to the total number of buffer positions. The number of tokens on place F indicates the current number filled buffer positions.
Initially F has zero tokens
M has only one token and it is in the input place for both the transitions t2 and t3. So either t2 or t3 fires at any particular time. In this way M prevents simultaneous insertion and removal of items.

For the transition t2 (place an item in the bufer) E belongs to the set of-input-place and F belongs to the set of output place. That means t2 will be able to fire only if there is atleast one empty buffer position and on firing t2 produces one filled buffer positions. For the transition t3 (Remove an item from the buffer) F acts as the input place and E acts as the output place. The transition t3 will be able to remove an item only if there is atleast one filled buffer position. After removing an item the empty buffer positions is incremented by 1. E prevents buffer overflow, F prevents underflow and M prevents simultaneous reading and writing of the buffer. Various properties of Petri nets can be determined using analysis techniques based on the notion of net invariants. An invariant is a set of places whose total number of tokens remains constant and which has no invariat subsets. In fig 3.20, the invariants are {P1,P2}, {P3,P4},{P5,P7} and {Po}. The net in fig 3.20 is "bounded" because each place is in some invariant, and the net has N+3 total tokens. The net is "conservative" because the invariants are disjoint and inclusive. Thus, N+3 is a constant number of tokens. The petri net in fig 3.20 exhibits mutual exclusion between T2 and T3 because both are marked by P6 and {P6} is an invariant, having one token. Thus, t2 and t3 are in conflict and mutually exclusive. The buffer cannot underflow because {P5,P7} is an invariant with N tokens. Thus M(15) +M(P7) = N at all times. If M(P5) =N. then the buffer is empty ie M(P7)+N implies M(P5) = 0, and the buffer cannot overflow.

Deadlock is not possible in the net of fig 3.20 in a deadlocked net, no transition can fire. Assume that the net is deadlocked. Then in particular T2 cannot fire. Thus, M(P5) = 0 or M(P2) = 0 . Assume that M(P2) =0 then M(pl) = 1 and tl can fire because {P1.P2} in an invariant having one token. Now assume that M(P5) =0. This implies M(P7) = N because {P5,P7J is an invariant with N tokens. If M(P7)+N and M(p3) =1,T3 can fire. If M(P3) =0 then M(P4) =1 because {P3,P4} is an invariant. If M(P4) =1 then T4 can fire. If T2 cannot fire, we have proven that Tl or T3 or T4 can fire. Thus, the net in fig.4.13 cannot deadlock.

Bounded, conservative Petri nets are equivalent to finite state automata. In a finite-state automation representation of a petrinet, the state of the net is given by the marking configuration, and transitions leading to next states change the markings to reflect changes in the next state. The equivalence of bounded, conservative Petri nets to finite-state automata makes all the analysis tools for finite state automata available for their class of petri nets. Although, bounded, conservative Petri nets are equivalent to finite-state automata. Petri nets provide a more convinient notation for specifying and analysing concurrent systems, Inbounded and non-conservative Petrinets provide more powerful mechanisms than do finite automata,

Languages and processors for requirements specification.

A number of special-purpose languages and processors have been developed to permit concise statement and automated analysis of requirements specifications for software. Some specfcation languages are graphical in nature. While others are textual, all are relational in nature some specification languages are manually applied and others have automated processors. In this section we provide a brief introduction to few specification languages and processors.

3.7.1.PSL/PSA

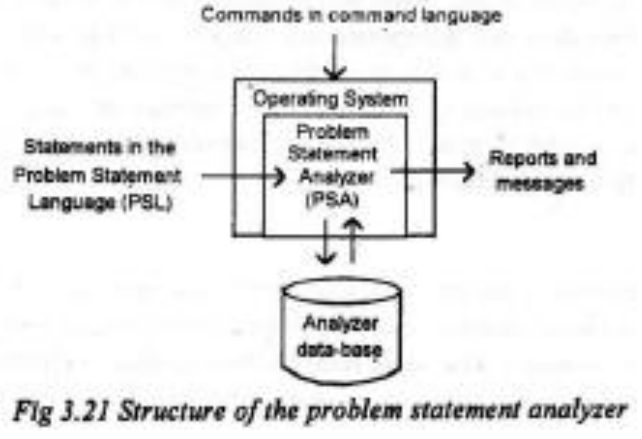
The Problem Statement Language (PSL) was developed by Professor Daniel Teichrow at the University of Michigan, The Problem Statement Analyser (PSA) is the PSL processor. PSL is based on a general model of systems. This model describes the system as a set of objects, where each object may have properties and each property may have property values. Objects may be inter-connected; the connections are called relationships. The general model is specialized to

information systems by allowing only a limited number of predefined objects, properties and relationships.
In PSL, system descriptions can be divided into eight major aspects:
1. System input/output flow.
2. System structure
3. Data structure *A,*
4. Date derivation
5. System size and volume
6. System dynamics
7. System properties
8. Project management

The system input/output flow aspects deals with the interaction between a system and its environment. System structure is concerned with the hierarchies among objects in a system. The data structure aspect includes all the relationships that exist among data used and /or manipulated by a system as seen by the users of the system. Data derivation describes data relationships that are internal to a system. The system size and volume aspects is concerned with the size of the system and those factors that influence the volume of processing required. The system dynamic aspects of a system description presents the manner in which the system "behaves" overtime. The project management aspect requires that project-related information, as well as product-related informations be provided. This involves identification of the people involved, their responsibities, schedules, cost estimates etc.

The Probem Statement Analyser (PSA) is an automated analyzer for processing requirements stated in PSA. The structure of PSA is an automated analyzer for procesing requirements stated in PSA is illustrated in fig 3.21 PSA operates on a data base of information collected from a PSL description the PSA system can provide reports in four categories.
1. Data-base modification reports
2. References reports
3. Summary reports
4. Analysis reports

Data-base modification reports list changes that have been made since the last report together with diagnostic and warming messages, these reports provide a record of changes for error correction and recovery. Reference reports include.
a. Name list report
b. Formatted problem statement Report and
c. Dictionary Report
Name List Report lists all the objects in the database with types and dates of last change. The Formatted problem Statement Report shows properties and relationships for particular object. The dictionary report nrovides a data dictionary.


Commands in command language
Operating System
Problem Statement Analyzer (PSA)
Statements in the Problem Statement Language (PSL)
Reports and messages
Analyzer data-base
Fig 3.21 Structure of the problem statement analyzer

Summary reports present information collected from several relationships. These include
a. Data Base Summary Report
b. Structure Report and
c. External Picture Report

The Data Base Summary Report provides project management information by listing the total number of objects of various types and how much has been said about them. The structure Report shows

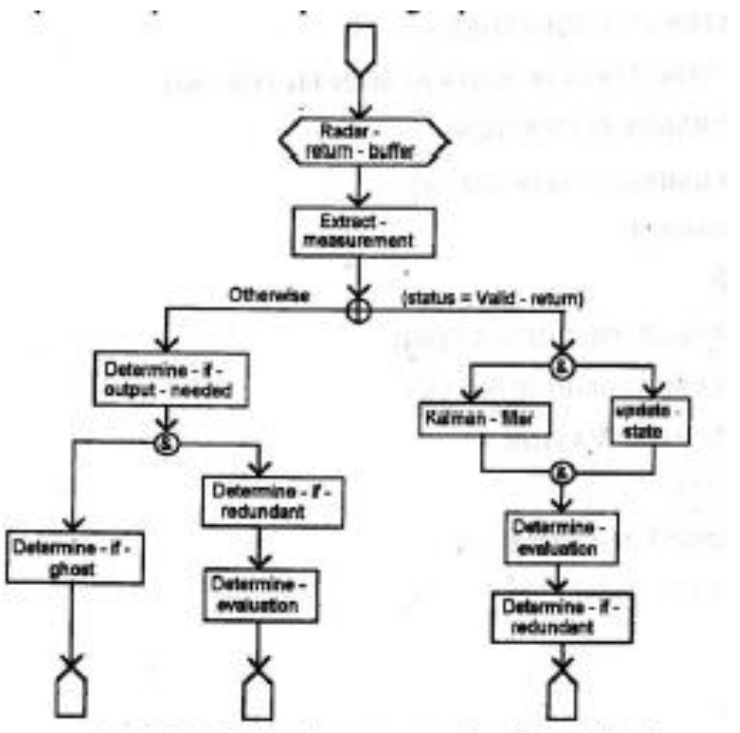


Fig 3.22 Graphical representation of an R-NET.

complete and partial hierarchies and the External picture Report depicts data flows in graphical form.
Analysis reports include
a. Contents Comparison Report
b. Data Processing Interaction Report
c. Processing chain Report.

The contents comparison Report compares the similarity of inputs and outputs. The data processing Interaction Report can be used to detect gaps in information flow and unused data objects. The processing chain Report shows the dynamic behaviour of the system.

PSL/PSA ia a useful tool for documenting and communicating software requirements. Tools do not solve problems; people solve problem. Software tools support the problem solving that is done by humans. Good software tools are properly used can significantly improve software quality and programmer productivity. PSL/PSA not only supports requirements analysis, it also supports design. The PSL user may sometimes have the tendency to get involved in too many design details before highlevel requirements are completed. PSL/PSA does not incorporate any specific problem solving techniques. PSA does provide the capability to define new PSC constructs and new report formates, which allows the system to be tailored to specific problem domain and specific solution methods. PSL/PSA has been used in many different situation ranging from commencial data processing appliction to air defense systems,

RSL/REVS:

The Requirements Statements language (RSL) was developed by the TRW Defense and Space systems Group to permit concise^ unambiguous specification of requirements for real-time software systems. The Requirements Engineering Validation System (REVS) processes and analyzes RSL statements both RSL and REVS are components of the
100

Software Requirement Engineering Methodology (SREM).

Many of the concepts in RSL are based on PSL. For example, RSL has four primitive concepts.

1. Elements, which name objects
2. Attributes, which describe the characteristics of elements.
3. Relationships, which describe binary relations between elements and
4. Structure which are composed of nodes and processing steps "Data" is an example of RSL element. "Initial Value" is an attribute of the element. Data and Input specifies a relationship between a data item and a processing step.

In RSL flow oriented approach is used to describe real-time systems. RSL models the stimulus-response nature of process control-systems each flow originates with a stimulus and continues to the final response. Specifying requirements in this fashion makes explicit the sequences of processing steps required. A processing step may be accomplished by several different software components and or software component may incorporate several processing steps. A sequence of processing steps may involve hardware, software and people components. Performance characteristics and validation conditions can be associated with particular points in the processing sequence.

Flow are specified in RSL as requirements networks (R-NETS). R-NETS have both graphical and textual representations as illustrated in fig 3.22 a and b R-NETS can be used to specify parallel (order-independent) operatoins using the AND node (&). Fan-in at the end of an AND^{J} structure is a synchronization point; all parallel paths must be completed before any subsequent processes are initiated. The OR node (+) has a condition associated with each path. Each OR node must have an otherwise path that is processed if none of the conditions are true. If mor^ than one condition is true, the first path as indicated by either implicit or explicit ordering is processed. The third structure type is the For-each, which contains only one path. The path is processed once for each element in a specified set of system entities. Iteration over the set is not implied to be in any particular order.

```
R-NET: PROCESS-RADAR-RETURN
STRUCTURE
INPUT-INTERF ACE-RADAR-RETURN-BUFFER
EXTRACT-MEASUREMENT
DO(STATUS - VALIDRETURN)
DO UPDATE-STATE AND KALMAN-FILTER END
DETERMINE ELEVATION
DETERMINE-IF-REDUNDANT
        TERMINATE
OTHERWISE
DETERMINE-IF-OUTPUT-NEEDED
DO      DETERMINE-IF-REDUNDANT
DETERMINE ELEVATION TERMINATE
AND            DETERMINE-IF-GHOST
TERMINATE
END
END
END
```

Fig 3.22 textual representation of an R-NET

RSL incorporates a number of predefined element-types, relationships, attributes and structures. Predefined elements include Alpha, Data and R-NET. An Alpha specifies the functional characteristics of a processing step in an R-NET. Alphas are described in terms of other elements such as Inputs. Outputs and Descriptions.

A Data Elements specifies data elements at the conceptual level, using relationships such as Input-to and Output-from attributes such as Initial Value and Includes and elements such as Description. Fig. 3.23 illustrates two Alphas one Data and one Originating-Requirement.

In addition to the predefined elements, types, relationships and attributes in RSL new elements relationships and attributes can be added to the language using Define. Fig 3.24 illustrates the definition of two new elements. Input-Interface and Messsage and the definition of their relationships. Parsers A complete definition would include relationships to associate particular items of Data with messages. After the new items have been defined, they can be used to specify other attributes of the system.

ALPHA: EXTRACT-MEASUREMENT
INPUTS: CORRELATED-RETURN
OUTPUTS: VALID-RETUN.MEASUREMENT
DESCRIPTION: "DOES RANGE SELECTION PER CISS REFERENCE 2-7"
ALPHA: DETERMINE-IF-REDUNDANT . INPUTS: CORRELATED-RETURN
OUTPUTS: REDUNDANT-IMAGE
DESCRIPTION: "THE IMAGE OF THE RADAR RETURN IS ANALYZED TO DETERMINE IF IT IS REDUNDANT WITH ANOTHER IMAGE"
DATA : MEASUREMENT
INCLUDES: RANGE-MARK-TIME.AMPLITUDE.RANGE-VARIANCE
RD-VARIANCE. R-AND-RD-CORRELATION
OUTPUT FROM ALPHA EXTRACT-MEASUREMENT
DESCRIPTION: "THIS IS THE ESSENCE OF THE INFORMATION IN THE RETURN"
ORIGINATION-REQUIREMENT
DPSPR-3 2.2 A-FUNCTIONAL
DESCRIPTION "ACTION SEND RADAR ORDER INFORMATION
INFORMATION RADAR ORDER IMAGE(REDUNDANT)" TRACES TO
ALPHA COMMAND-PULSES ALPHA DETERMINE-IF-REDUNDANT
MESSAGE RADAR-ORDER-MESSAGE DATA REDUNDANT-IMAGE
ENTITY-CLASS IMAGE *Fig 3.2) Examples ofALPHA.DATA and ORIGINATING-REQUIREMENT*
DEFINE ELEMENT-TYPE INPUT-INTERFACE
(* A port between the data processing system and the rest of the system which aspects data from another part of the system*) }
DEFINE ELEMENT-TYPE MESSAGE
(* An aggregation of DATA FILES that PASS through an interface as a logical unit *)
DEFINE RELATIONSHIP PASSES
(* An INPUT-INTERFACE "PASSES" A logical aggregation of data called a MESSAGE from the outside system into the data processing system *)
COMPLEMENTARY RELATIONSHIP PASSED-BY
SUBJEC1 INPUT-INTERFACE
OBJECT MESSAGE
Fig 3.24 Examples of the DEFINE attribute from RSL

The Requirements Engineering and Validation System (REVS) operates on RSL statements. REVS consists of three major components.

1. A translator for RSL.
2. A centralised data base, the Abstract System Semantic Model (ASSM)
3. A set of automated tools for processing informations in ASSM.

A schematic diagram of REVS is presented in fig 3.25 ASSM is a relational data base similar in concept to the PSL/PSA data base. Automated tools for processing information in the specifying flow paths, static checkers that check for completeness and consistency of the information used throughout the system and an

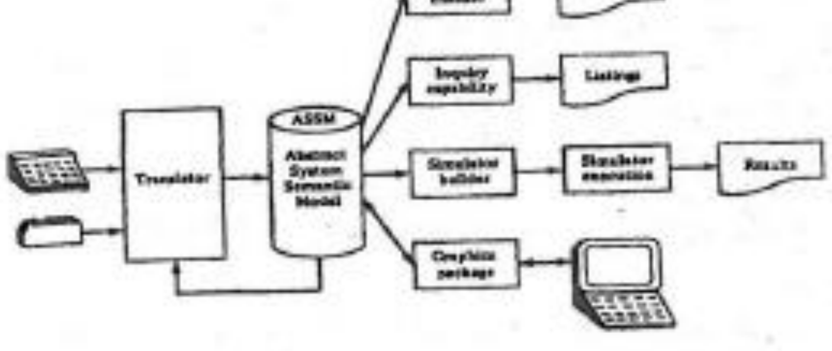


Fig. 3.25 Structure of the REVS processor

automated simulation package that generates and executes simulation models of the system In addition to standard reports and displays. REVS provides a capability for defining specific analysis and report that may be needed on particular projects.

REVS is a large, complex software tool. Use of the REVS system is cost effective only for specification of large, complex real-time systems. The RSL notation can be manually applied to specify real-time system characteristics.

3.7.2. Structured Analysis and Design Techniques (SADT)

SADT incorporates a graphical language and a set of methods and management guidelines for using the language. The SADT language is called the language of Structured Analysis (SA).

An SADT model consists of an ordered set of SA diagrams. Each diagram is drawn on a single page and each diagram must contain three to six nodes plees inter connecting arcs. Two basic types of SA diagrams are the activity diagram (actigram) and the data diagram (datagram). On an actigram the nodes denote activities and the arcs specify data flow between activities on the arc. Actigrams and datagrams are thus duals. In practice, activity diagrams are used more frequently thari data diagrams; however data diagrams are important for atleast two reasons: to indicate allabtivities affected by a given data object, and to check the completeness and consistency of an SADT model by constructing data diagrams from a set of activity diagrams.

Fig a and b illustrates the formats of actigrams and datagram nodes.

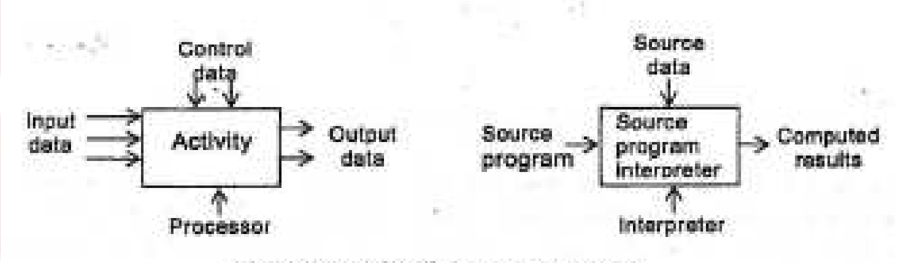


Fig.3.26 Activity diagram components.

Arcs coming into the left side of a node carry inputs and arcs leaving the right side of a node-convey outputs. Arcs entering the top of a node convey control and arcs, entering the bottom specify mechanism. Outputs provide input and control for other nodes. Outputs from some nodes are the system output to the external environment. Similarly, input and control must come from the output of other nodes or from the external environment. Input, control and output may be connected to nodes on other pages in a set of diagrams. Connecting lines to other diagrams are indicated by a composite diagram/line number.

In an activity diagram the inputs and outputs are data flows and the mechanisms are processes. Control is data that is used, but not modified by the activity.

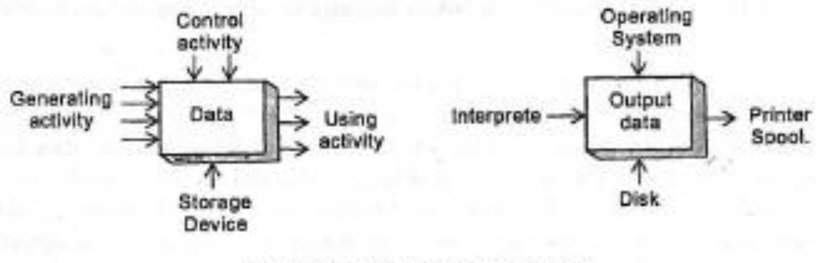


Fig.3.26 Data diagram components

g 4.20- illustrates some structural characteristics of SA diagrams

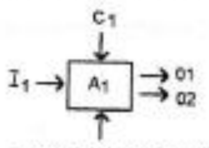


Fig 3.27 Activity A1

In a data diagram the input is the activity that creates the data object and the output is the activity that uses the data object. Mechanisms in data diagrams are the devices used to the representations of data object. Mechanisms in data diagrams are the devices used to the representations of data objects. Control in a data storegram controls the conditions under which the node will be activated. In both actigrams and datagrams controls are provided by the external environment and the output of other nodes.

In fig 3.27 II is the external input to the diagram and Cl is an external control. The output of activity All is input to activities A12,A13 and A14. Activities A12 and A14 can proceed in parallel but activity A13is controlled by the output of A12 and must wait for that output. The output of activity A13 feeds back to control activity Al 1. Outputs from the diagram are 01 and 02. Mechanisms are indicated by Ml through M4. As illustrated in tig.3.27 each node in an SA diagram can be expanded into other diagrams that are titled and numbered to indicate the hierarchical relationships among diagrams. External inputs outputs, controls and mechanisms for a diagram that is a decomposition of a node in another diagram are restricted to the inputs, outputs, controls and mechanisms of the node being decomposed. Further, all the arcs incident on the node in the parent diagram must be depicted in the expanded diagram.

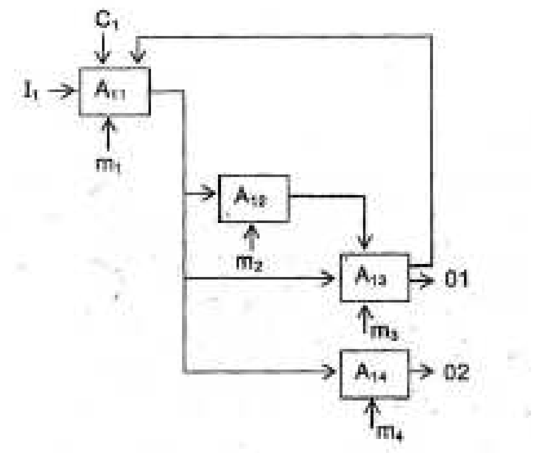


Fig.3.27 An expanded view of activity A1

The SA language has a large number of features for indicating aspects such as bounded context, necessity, dominance, relevance, exclusion, interfaces to parent diagram, unique decompositions, shared decomposition and cooperation. A primary goal for the language is to provide a medium in which a wide variety of complex situations can be specified using decomposition and structural relationships.

Management techniques for developing an SADT model include author-reader cycles for reviewing the diagrams, and a project librarian, who assigns document numbers and controls the flow of documents.

The SADT methodology provides a notation and a set of techniques for understanding and recording complex requirements in a clear and concise manner. Major strengths of SADT include the top-down methodology inherent in decomposing high-level nodes into subordinate diagrams, the distinction between input, output, control and mechanism for each node, the duality of activity and datagrams and the management techniques for developing reviewing and coordinating an SADT model. SADT can be applied to all types of systems; it is not limited to software applications.

3.7.3. Structured System Analysis (SSA)

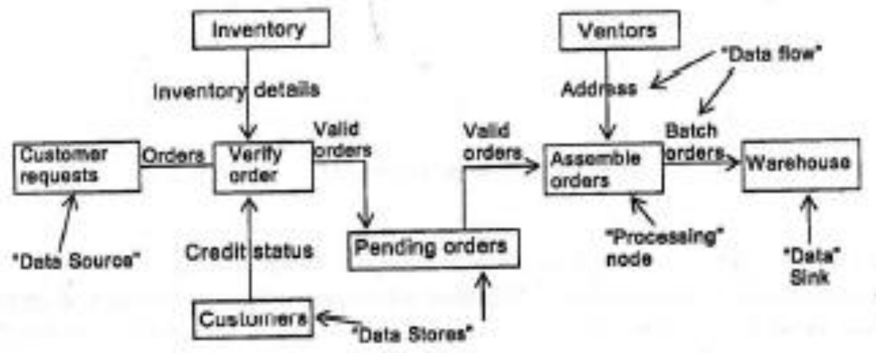


Fig 3.28 A Data flow diagram

Structured System Analysis (SSA) is used primarily in traditional data processing environments. SSA uses a graphical

language to build models of systems. There are four basic features in SSA.
data flow diagrams
data dictionaries
procedure logic representation
and data store structuring techniques.
SSA data flow diagrams are similar to SADT actigrams, but they do-not indicate mechanism and control and an additional notation is used to show data stores. An SSA data flow diagram is illustrated in fig 3.28

In fig 3.28 open ended rectangles indicate data stores, labels on the arcs denote data items, shaded rectangles depact source and sinks for data and the .remaining rectangles indicate processing steps.

A data dictionary is used to define and record data elements. A typicl data dictionary entry is presented in fig 3.29.

DATA FLOW: ORDER
COMPOSITION:
CUSTOMER-IDENTITY
ORDER-DATE
ITEM-ORDERED+
CATALOG-NUMBER
ITEM-NAME
UNITPRICE
QUANTITY
TOTAL-COST
DATA STORE: CUSTOMER-IDENTITY
COMPOSITION
NAME
FIRST
MIDDLE
LAST
PHONE
NUMBER
EXTENSION (Optional)
ADDRESS
STREET
CITY
STATE DATA STORE: ORDER-
DATE COMPOSITION
TIME
DAY
MONTH
YEAR

Fig 3.29 Data dictionary entries in SSA

NOTE: The "+" Notation denotes one or more occurences of the ITEM-ORDERED field.

Processing logic representation, such as decision tables and structured English are used to specify algorithmic processing details. Processing logic representations are used to precisely specify processing sequences in terms that are understandable to customers and developers. A processing logic representation is illustrated in fig 3.30

INITIALIZE the program (open files, initialize counters and
tables) READ the first order record
WHILE there are more item-ordered fields in the order
record do EXTRACT the next item-ordered SEARCH the

```
order-table for the extracted item IF the extracted item is
found THEN
    INCREMENT the extracted item's occurence count
ELSE
INSERT the extracted item into the order-table and
INCREMENT the occurance count ENDIF
    INCREMENT the items-processed counter END
WHILE at the end of order record END WHILE when
all order records have been processed WRITE the
order-table and the items-processed counter TO the
batch-order file CLOSE files TERMINATE the
program
```

Fig 3.30 An SSA processing Logic representation

As illustrated in the above figure, an SSA specification can be refined to the level of detailed design. As with PSL, there is danger that programmers using SSA will proceed to the detailed design level for some processing nodes before the data flow diagram and data dictionary are completed. This may be advisable for certain critical nodes, but as a general rule it is better to defer detailed design considerations until analysis is complete.

Two similar versions of SSA have been described by Gane and Sarson and by DeMarce. The Gane Sarson version incorporates the data-base concepts. An important features of SSA, as presented by Gane and Sarson, is the use of a relational model to specify data flows and data stores. Relational are composed from the fields of data records. The fields are called the domains of the relation. If a record has N fields, the corresponding relation is called an N tuple. For example, the ORDER record is fig 4.22 can be represented by the 7 tuple.

ORDER("CUSTOMER-IDENTIFY.ORDER-DATE CATALOG-NUMBER:, ITEM-NAME, UNIT-PRICE.QUANTITY. TOTAL-COST)

This relation is in "first normal form because there are no repeating groups in the relation. The ORDER record in fig 4$_f$22 was converted to first normal form by associating CUSTOMER - ITENDITY and ORDER-DATE with each individual item ordered. The first three domains of the relation are quoted to indicate that they form the key for unique identification of each order.

The relation must be transformed from first normal form to second normal form and second normal form to third normal form which is the simplest one. To transform the relation from the first normal form to second normal form all non-key domains must be made to fully functionally dependent on the key. In this example ITEM _NAME and UNIT-PRICE are dependent only on the CATALOC-NUMBER portion of the key, they are not fully dependent on the entire key. On the other hand the QUANTITY cannot be determined without knowing the entire key. Thus, quantity is fully functionally dependent on the key, but ITEM-NAME and UNIT-PRICE are not.

The partial dependence of ITEM-NAME and UNIT-PRICE can be eliminated by placing the domains that describe the ordered item in a seperate relation. This results in the following second normal form.

. _ ORDER ("CUSTOMER - IDENTITY, ORDER - DATE, CATALOG - NUMBER" QUANTITY.TOTAL-COST)

ITEM("CATALOG-NUMBER."ITEM-NAME.UNIT-PRICE)

In this, ORDER is having two non key domains QUANTITY and TOTAL-COST which are fully dependent on the entire key and in ITEM, the non-key demains ITEM-NAME and UNIT_PRICE are dependent on the CATALOG-NUMBER which is the key for ITEM.

To obtain third normal form from second normal form, we must ensure that every non-key domain is functionally independent of all other non-key domains. In this example, TOTAL-COST can be computed from QUANTITY and UNIT-PRICE. Therefore TOTAL-COST is eliminated to produce two relations that are both in third normal form.
ORDER("CUSTOMER-ID ENTITY.ORDER-DATE.CATALOG-NUMBER",QUANTITY)
ITEM("CATALOG-NUMBER," ITEM-NAME, UNIT-PRICE)

This is the simplest and most desirable form of normalization. The unwieldy three -domain key in the ORDER relation can be eliminated by adding a new key domain called ORDER-NUMBER to produce the relation.

ORDER("ORDER-NUMBER."CUSTOMER-IDENTITY. ORDER-DATE, CATALOG-NUMBER, QUANTITY)

This has the advantage of simplifying access to orders, but it also requires that the order number be known before an order relaton can be accessed.

3.7.4. GIST

Gist is a textual language based on a relational model of objects and attributes. A Gist specification is a formal description of valid behaviours of a system.
A specificaton is composed of three parts.
1. A specification of object types and relatonships between these types. This determines a set of possible states.
2. A specification of actions and domains which define transistions between possible states.
3. A specification of constraints on states and state transitions.
The valid behaviours of a system are those transition sequences that do not violate any constraints.
Preparng a Gist specification involves several steps.
1. Identify a collection of object types to describe the objects manipulated by the process types are typically described by concrete nouns that convey meaningful information.
2. Identify individual objects (values) within types. These individuals are not variables in the progrmming language sense, they may be used to describe constraints or to describe dynamic aspects of a process. Often a specification will have no individual objects.
3. Identify relationships to specify the ways in which objects can be related to one anothers. The nature of relationships is highly dependent on the process being specified. Typical relationships include "connects to", "is part of and "derived from"
4. Identify types and relationships that can be defined (possibly by recursive definition) in terms of other types and relation. These situations provide the basis for derivation rules. Derivation rules permit expansion of the specification vocabulary.
5. Identify static constraints on the types and relationships static constraints identify processing steps that should never arise. Static constraints occur because of the physical reality being modeled and because the derived process may restrict potential states.
6. Identify actions that can change the state of the process in some way. For each action, the specification should state the types of objects on which the action can operate, any precondition on such objects that restrict the situation in which the action can be performed and a statement of how the action changes the process state.
7. Identify dynamic constraints They consist of conditions that restrict allowable changes in the process state. Typical dynamic constraints include restrictions prohibiting changes in certain relationshsips between objects following their creation and restrictons on the nature of change in relations.

8. Identify active participants in the process being specified and group them into classes such that common actions are formed by the participants in a class.
fig 4.24 illustrates the use of types, relations, constraints and action in a Gist specification
type SHIP includes (USS-CONST-HMS-QM2);
type CARGO definition (GRAIN,FUEL);
type TONNAGE definition natnum;
relation CONTAINS (SHIP,CARGO,TONNAGE) where always prohibited (SHIPS,FUEL,GRAIN)//
CONTAINS(SHIP,FUELS,$)
CONTAINS(SHIP.GRAIN,$)
action**LOADSHIP(SHIP,CARGO,INCR: TONNAGE)**
precondition SHIP: DOCK.HANDLESS = CARGO.
*il*definition if CONTAINS (SHIP,CARGO.$)
then update TONNAGE
Fig 3.31 A Gist specification (adapted from BAL 81)

The specification identifies types ship, cargo and tonnage ships include individuals such as the U.S.S. Constitution and the H.M.S. Queen Mary. Cargoes are grain and fuel Tonnage is a natural number; natnum is a predefined type in Gist. The relation
"CONTAINS" in fig 3.31 states that ships contain cargoes of specified tonnage for example CONTAINS(USS-CONST.GRAIN.50) means that the USS Constitution contains 50 tons of grain. CONTAINS is restricted to prohibit simultaneous cargoes of grain and fuel. Action "LOADSHIP" describes the change in process state that occurs when a cargo is added to a ship. The precondition states that the ship must be docked at a pier that handles the specified cargo.

The Initial operating capability (IOC) is a prototype testing facitlity for software specifications written in Gist. The purpose of IOC is to validate functional specifications by "executing" them on real test data. IDC consists of an evaluator capable of executing specification expressed in a subset of Gist and programs that permit entering editing and displaying of Gist specification. In addition, IDC permits state initialization, displaying of states and interactive break pointing and tracing of test case evaluatidn.

Gist has a well-defined but rather complex syntax. Learning the syntax of Gist is similar to learning the syntax of a new programming language. Learning to use Gist is complicated by the fact that one is not only learning a new notation , but also learning a new way of thinking about system and learning new techniques for specifying functional behaviour.

Primary application areas for the notation and tools discussed in the section are summarized as follows.
PSL/PSA : Originally developed for data processing application. Widely used in other applications
RSL/REVS : Real-time process control systems
SADT • .Interconnection structure of any large, complex system. Not restricted to software systems.
SSA :. Gane and Sarson version used in data processing applications that have database requirements.
DeMarco version suited to data flow analysiss of software systems
Gist : Object-oriented specification and design Refinement of specifications un to source code.

Unit 3: Software Design:

Introduction

According to webster, the process of design involves" conceiving and planning out in the mind" and "making a drawing, pattern, or sketch of. In software design, there are three distinct types of activities: external design, architectural design and detailed design. Architectural and detailed design are collectively referred to as internal design.

External design of software invloves conceiving planning out and specifying the externally observable characteristics of a software product. These characteristics include user displays and report formats, external data sources and data sinks and the functional characteristics, performance requirements and high level process structure for the product. External design begins during the analysis phase and continues into the design phase. In practice it is not possible to perform requirements definition without doing some preliminary design. Requirements definition is concerned with specifying the external, functional and performance requirements for a system, as well as exception handling. External design is concerned with refining those requirements and establishing the high level structural view of the system.

Internal design involves conceiving, planning out and specifying the internal structure and processing details of the software product. The goals of internal design are to specify internal structure and processing details, to record design decisions and indicate why certain alternatives and trade-off were chosen, -to elaborate the test plan, and to provide a bluprint for implementation, testing and maintanance activities. ..

Architectural design is concern with refining the conceptual view of the sytem, identifying internal processing functions, decomposing high-level functions into sub functions, defining internal data streams and data stores. Issues of concern during detailed design include specification of algorithms that implement the functions, concrete data structures that implement the data stroes, the actual interconnections among functions and data structures and the packaging scheme for the system.

The test plan describes the objectives of testing, the test completion criteria, the integration plan, particular tools and technique to be used and the actual test cases and excepted results. Functional tests and performance tests are developed during requirements analysis and are refined during the design phase. Tests that examine the internal structure of the software product and tests that attempt to break the system (stress tests) are developed during & implementation.

Fundamental Design concepts

Every intellectual discipline is characterized by functional concepts and specific techniques. Techniques are the manifestations of the concepts as they apply to particular situations. Techniques come and go with changes in technology, intellectual fads, economic conditions and social concerns. By definition fundamental principles remain the same .throughout. They provide the underlying basis for development and evaluation of techniques. Fundamental concepts of software design include abstraction, structure, information hiding, modularity, concurrency, verification, and design aesthetics.

4.1.1. Abstraction

Absraction is the intellectual tool that allows us to deal with concepts apart from particular instances of these concepts. During requirements definition and design, abstraction permits separation of the conceptual aspects of a system from the implementation details. We can, for example, specify the FIFO property of queue or the LIFO property of a stack without concern for the representation scheme to be used in implementing the stack or queue. Similarly, we can specify the functional characteristics of the routines that manipulate data structures ((e.g)

NEW, PUSH, POP, TOP, EMPTY) without concern for the algorthmic details of the routines.

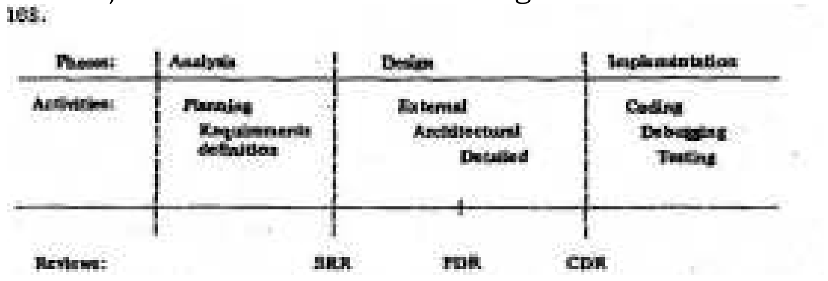


Figure 4.1 Tuning of die Software Requirements Review (SRR), the Preliminary Design Review (PDR), and the Critical Design Review (CDR).

During Software Design, abstraction allows us to organize and channel' our through processes by postponing structural considerations and detailed algorthmic considerations until the functional characteristics, data streams, and data stores have been established.

Three widely used abstraction mechanisms in software design are functional abstraction, data abstraction and control abstraction. Functional abstraction involves the use of parameterized subprograms. The ability to parameterize a subprogram is a powerful abstraction mechanism. Functional abstraction can be generalised to collections of subprograms.

Data abstraction involves specifying a data type or a data object by specifying legal operations on objects; representation and manipulation details are suppressed.

The term "data encapsulation" is used to denote a single instance of a data object defined in terms of the operations that can be performed on it; the term "abstract data type" is used to denote declaration of a data type (such as stack) from which numerous instances can be created.

Abstract data type are abstract in the sense that representation details of the data items and implementation details of the functions that manipulate the data' items are hidden within the group that implements the abstract type. During architectural design, the visible functions that operate on abstract objects are specified. Specification of representation details for data objects and algorithmic.details for the functions are postponed until detailed design.

It should not be inferred from over simple examples that data abstraction is only useful to specify simple data items such as stacks and queues. The contents of a graphics display screen or a run time activation record for a block structured programming language might, for example, be an object of abstract type is defined in terms of permitted operations on that object. An abstract data type can be defined in terms of other abstract types. Instances of stack can be created as

```
Package STACK-TYPE is
Type STACK is limited private; Procedure PUSH (I: in INTEGER;
Sr in out STACK); Procedure POP (I: OUT in INTEGER; S: in out
STACK); Function EMPTY (S: in STACK) return BOOLEAN;
Function FULL (S: in STACK) return BOOLEAN;
STKFULL,STKEMPTY: exception; Private ,
type STACK is record .
INDEX: INTEGER range 0 100 := 0;
STORE: array (I... 100) of INTEGER;
end record;
end STACK-TYPE
STACK1 .STACK2:STACK;
PUSH(X,STACK1); '
POP(Y,STACK2);
B:=EMPTY(STACK1);
```

Control abstraction *is* the third commonly used abstraction mechanism in software design. Control abstraction is used to state a desired effect without stating the exact mechanism of control. If statements and WHILE statements in modern programming languages are abstractions of machine code implementations that involve conditional jump instructions. A statement of the form "For all I in S sort files 1" leaves unspecified the sorting technique, the nature of the files and how "For all I in S" is to be handled.

4.1.2. Information Hiding

Information hiding is a fundamental design concept for software, When a software systems is designed using the informations hiding approach, each module in the system hides the internal details of its processing activities and module communicate only through well-defined interfaces.

In addition to hiding of difficult and changeable design decisions, other candidates for information hiding include.

1. A data structure its internal linkage, and the implementation details of the procedures that manipulate.
2. The format of control blocks such as these for queues in an' operating system.
3. Character codes, ordering of character sets and other implementations details.
4. Shifting, masking and other machine dependent details:

4.1.3. Structure

Structure is a fundamental characteristic of computer software. The use of structuring permits decomposition of a large system in to smaller, more manageable units with well-defined relation ships to the other units in the system. The most general form of system structure is the network. A computing network can be represented as a directed graph, consisting of nodes and arcs. The nodes can represent processing elements that transform data and the arcs can be used to represent data links between nodes. Alternatively, the nodes can represent data stores and the arcs data transformation.

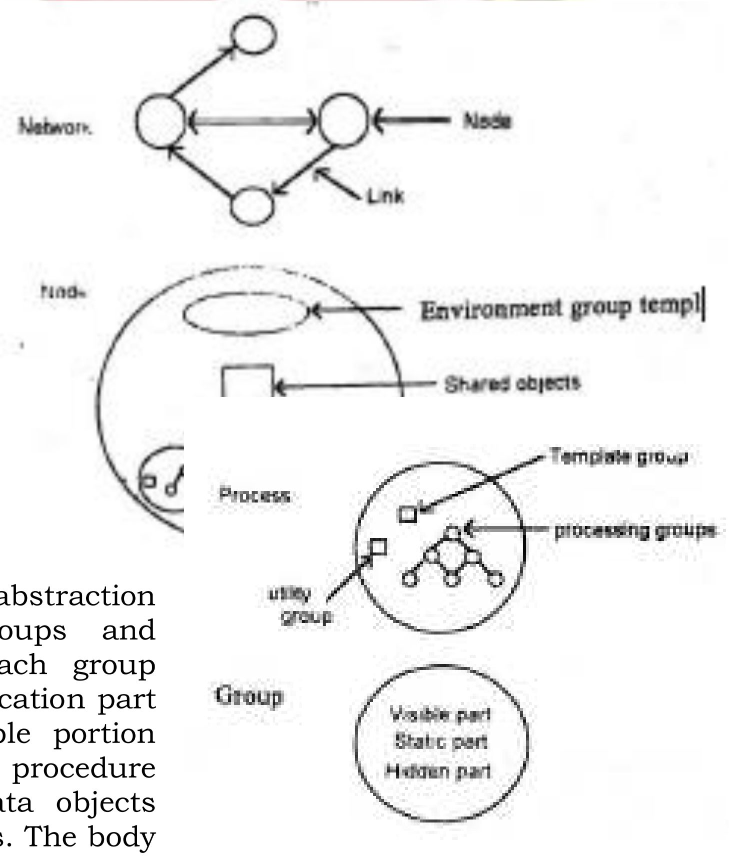


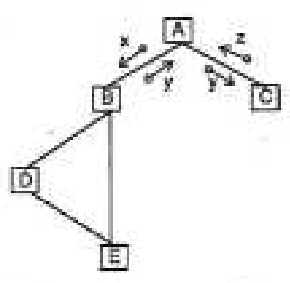
Fig. 4.1 Software System Structure

The structure inside a complex processing node moght consist of concurrent processes executing in parallel and communicating through some combinations of shared variable and synchronous and asynchronous message passing. Inside each processs, one might find funtional abstraction groups, data abstraction groups and control abstraction groups. Each group might consist of a visible specification part and a hidden body. The visible portion could provide attributes such as procedure interfaces, data types, and data objects available for use by other groups. The body of a processing group is typically a hierarchical collection of subgroups and Static data areas that implement the attributes specified is the visible specification part of the group.

The relationship "user" and the complementary relationship "is used by" provide the basis for hierarchical ordering of abstraction in a software system. The "user" relationship can be represented as a direct graph, where the notation A -> B means "A uses B" or "B is used by A".

Hierarchical ordering of abstraction is established by the following rule.
If A and B are distinct entities, and if A uses B then B is not permitted to use A or any entity that makes use of A.

A hierarchical ordering relations can be represented as an acyclic, directed graph with a distinguished node that represents the root entity. The root uses other entities, but it is not used by any entity. Subsidiary nodes in the hierachical structure represent subordinate entities that are used by their superordinates and inturn make use of their subordinates.
Hierarchical Structure may or may not form tree structures.
Note that in a tree there is a unique path from the root to each mode. In an ayclic, directed graph there may be more than one path from the root to a node.



1.2(a) graph structure chart

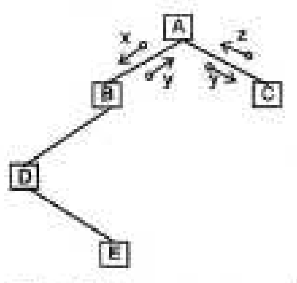


Fig.4.2(b) tree structure ch

The diagrams illustrated are called structure charts. Ther are not flowcharts because there is no indication of the processing sequence, and no indication of the conditions under which A might use B.

Directly recursive routines (those that invoke themselves) can be indicated on a structure chart by placing closed arcs on the modes for those routines. Indirectly recursive routines (A-> B -> C -> A) violate the hierchical structuring rule and should generally be avoided. Allowable exceptions include mutually recursive routines and coroutines (A <-> B).

A hierarchical structure isolates software components .and promotes ease of understanding, implementation, debugging, testing, integration and modification of a system.

4.1.4. Modularity

There are many definitions of the term "module". They range from "a module is a FORTRAN subroutine" to "a module is an Ada package" to "a module is a work assignment for an individual programmer". All of these definitions are correct, in the sense that modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction and each control abstraction handles a local aspect of the problem being solved.
Desirable properties of a modular system include.
1. Each processing abstraction is a^ell-defined subsystem that is potentially useful in other applications.
2. Each function in each abstraction has a single, well defined purpose.
3. Each function manipulates no more than one major data structure.
4. Functions share global data selectively. It is easy to identify all routine that share a major data structure.
5. Functions that manipulate instances of abstract data types are encapsulated with the data structures being manipulated.
Modularity enhances design clarity, which in turn eases, implementation, debugging, testing, documenting and maintanence of the software product.

4.1.5. Concurrency

Software systems can be categorized as sequential or concurrent. In a sequential system only one portion of the system is active at any given time. Concurrent systems hwe independent process that can be activated simultaneously if multiple processors are available. On a single processor, concurrent processses can be interleaved in execution time. This permits implementation of time shared, multiprogrammed and real-time systems.

Problems unique to concurrent systems include deadlock, mutual exclusion and synchronization processes. Deadlock is an undesirable situation that occur when all processes in a computing systems are waiting for other processes to complete some actions so that each can proceed. Mutual exclusion is necessary to ensure that multiple processors do not attempt to update the same components of the shared processing state at the same time synchronization is required so that concurrent processes operating at differing execution speeds can communicate at the appropriate points in their execution histories. Concurrency is a fundamental principle of software design because parallelism in software indroduces added complexity and additional degrees of freedom into the design process.

4.1.6. Verification / .

Verification is a fundamental concept on software design. Design is the bridge between customer requirements and an implementation that satisfies those requirements. A design is verifiable if it can be demonstrated that the design_ will result in an implementation that satisfies the customer's requirements this is typically done in two steps.

(1) verification that the software requirements definition satisfies the customer's needs.

(2) verification the the design satisfies the requirements definition.

4.1.7. Aesthetics

Aesthetic considerations are fundamental to design, whether in art or technology. Simplicity elegance and clarity of purpose distinguish products of outstanding quality from mediocre products. When we speak of mathematical elegance or structural beauty, we are speaking of those properties that go beyond more satisfaction of the requirements.

Modules and modularizing Criteria

Architechtural design has the goal of producing well structured modular software systems. A software module to be a named entity having the following characteristics.

1. modules contain instructions, processing logic and *daik* structures
2. modules can be seperately, compiled and stored in a library
3. modules can be included in a program.
4. module segments can be used by invoking a name and some parameter.
5. modules can use other modules.

Modularization allows the designes to decompose a system in to functional units to impose hierarchical ordering on function usage, to implement data abstractions, and to develop independently useful subsystems. In addition modularization can be used to isolate machine dependencies to improve the performance of a software product, or to ease debugging, testing, integration, tuning and modification of the system.

There are numerous criteria used, different system structures may result. Modulerization criteria include the conventional criteria in which each module and its submodules correspond to a processing step is the execution sequence, the information hiding criteria in which each module hides a difficult or chargeable design decision from the other modules, the data abstraction criteria in which each module hides the representation details of a major data structure behind functions that access & modify the data structure. Coupling-cohesion in which a system is structured to maximize the cohesion of elements **in** each module and to minimize **the** coupling **between** modules.

4.2.1. Coupling and cohesion

A fundamental goal of software design is to structure **the** software product so that the number and complexity of inter connection between modules is minimized.

The strength of coupling between two modules is influenced by the complexity of the interface, the type of connection and the type of communication. For example interfaces established by common control blocks, common data blocks common overlay regions of memory, common I/O devices and or global variable names are more complex that interface established by parameter lists parsed between modules.

Modification of a common data block or control block may require modificatons of all routines that are coupled to that block. If modules communicate only by parameters and if the interface between modules remain fixed, the internal details of modules can be modified without having to modify the routine that use the modified module.

Connection established by referring to other module names are more loosely coupled than connections established by referring internal elements of other modules. The degree of coupling is lowest for data communication, higher for control communication and highest for modules that modify other modules.
Coupling between modules can be ranked on a scale of strongest to weakest as follows.

1. content coupling
2. common coupling
3. control coupling
4. stamp coupling
5. data coupling

Content coupling occur when one module modifies local data values or instructions in another module. Content coupling can occur is assembly language programs.

Common coupling module are bound together by global data structures. For instance, common coupling results when all routines in a FORTRAN program reference a single common data block. <,

Control coupling involves passing control flags (as parameters or globals) between modules so that one module controls the sequence of processing steps in another module.

Stamp coupling is similar to common coupling, except that global data items are shared selecively among routines that require the data. Named common blocks in FORTRAN and packages in Ada support stamp coupling. Stamp coupling is more desirable than common coupling because fewer modules will have to be modified if **a** shared data structure is modified.

Data coupling involves the use of parameter lists to pass data items between routines. The most desirable form of coupling between modules is a combination of stamp and data coupling.

The internal cohesion of a module is measured in terms of the strength of binding of element with in the module. Cohesion of element occurs on the scale of weakest to strongest in the following order

1. coincidental cohesion
2. logical cohesion
3. temporal cohesion
4. communication cohesion
5. sequential cohesion
6. functional cohesion

7. information cohesion.

Coincidental cohesion occurs when the elements within a module have no apparent relationship to one another. This results when a large, monolythic program is "modularized" by arbitrarily segmenting the program into several small modules.

Logical cohesion implies some relationship among the elements of the module, as for example, in a module that performs all input and output operations or in a module that edits all data.

Module with temporal cohesion exhibit many of the same advantages as logically bound modules. However they are highly on the scale of binding because all elements are executed at one time and no parameters or logic are required to determine which elements to execute. A typical example of temporal cohesion is module that performs program initialization.

The elements of a module possessing communicational cohesion refer to the same set of input and or output data for example, "print and punch the output file" is cbmmunicationally bound. Communicational binding is higher on the binding scale than temporal binding because the elements are executed at one time and also refer to the same data.

Sequential cohesion of elements occurs when the output of one element is the input for the next element. For example, "Read Next transaction and Update Master File" is sequentially bound. Sequential cohesion is high on the binding sacle because the module structure usually bears a close resemblance on to the problem structure.

Functional cohesion is a strong and hence desirable type of binding of elements in a module because all elements are related to the performance of a single function. Example are"Computer Square Root"; "Obtain Random Number" and "Write Record to Output File".

Information cohesion of elements in a module occurs when module contains a complex data structure and several routines to manipulate the data structure. Each routine in the module exhibits functional binding. Information cohesion is the concrete realization of data abstraction. Information cohesion requires that only one functionally cohesion segent of the module be executed on each invocation of the module.

In summary, the goal of modularizing a software system using the coupling.Cohesion criteria is to produce systems that have stamp and data coupling between the modules and functional or informational cohesion of elements within each modules.

4.2.2. Other modularization criteria

Additional criteria for deciding which functions to place in which modules of a software system include hiding difficult and changeable design decisions, limiting the physical size of module structuring the system to improve observability and testability; isolating machine dependences to a few routines; easing likily changes; providing general purpose utility functions; developing an acceptable overlay structure in a machine with limited memory capacity; minimizing page faults in a virtual memory machine and reducing the call-return overhead of excessive subroutine calls. For each software product.the designer must weigh these factor and develop a consistent set of modularization criteria to guide the design process. Efficiency of the resulting implementation is a concern that frequently arises when decomposing a systems into modules. A large number of small modules having data coupling and functional cohesion implies a large execution time overhead for establishing run-time linkages between the modules.

The preferred technique for optimizing the efiiciency of a system is to first design and implement the system in a highly modular function. System performance in then measured and bottlenecks are removed by reconfiguring and recombining modules and by hand coding certain critical linkages and critical routine in assembly language if necessary.

Design Notations

In software design, as in mathematics, the representation schemes used are of fundamental importance. Good notation can clarify the interrelation ship and interactions of interest, while poor notation can complicate and interface with good design practice. Atleast three levels of design specification exist, external design specification, which describe the external characteristics of a software system; architechtural design specifications; which describe the structure of the system; and detailed design specifications, which describe control flow, data representation and other algorthmic details witfiin the module.
Notations used to specify the external characteristics, architechtural structure and processing details of a software system include data flow diagrams, structure chart. Hipo diagram procedure specifications pseudo code. Structural English, and structural flowcharts.

4.3.1. Data flow diagrams

Data flow diagrams ("bubble charts") are directed graphs in which nodes specify processing acitvities and the arcs specify data items transmitted between processing nodes. Like flowcharts, data flow diagram can be used at any desired level of abstraction. A data flow diagram might represent data flow between individual statements or blocks of statements in a routine, data flow between concurrent processes or data flow in a distributed computing system where each node represents a geographically remote processing unit unlike flowcharts, data flow diagrams do not indicate decision logic or conditions under which various processing nodes in the diagram might be activated.

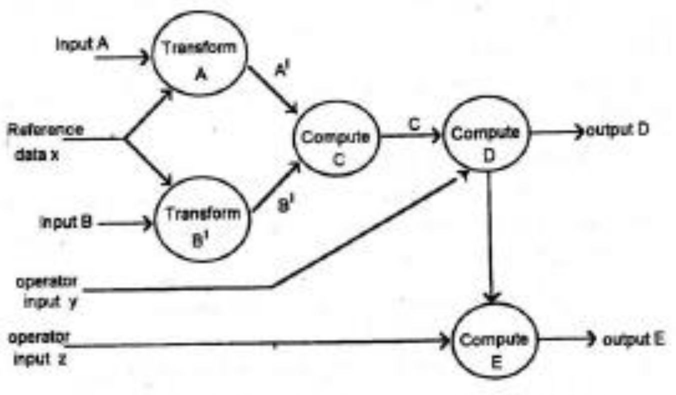


Fig: 4.3 An informal data flow diagram or "bubble chart"

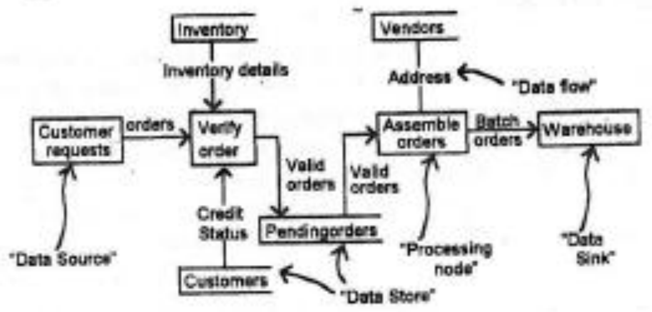


Fig: 4.4 A formal data flow diagram

Data flow diagrams are excellent mechanisms for communicating with customers during requirement analysis theyare also widely used for representation of external and top-level internal design specification.

4.3.2. Structure chart

Structure charts are used during architechtural design to document hierarchical structure, parameters and interconnection in a

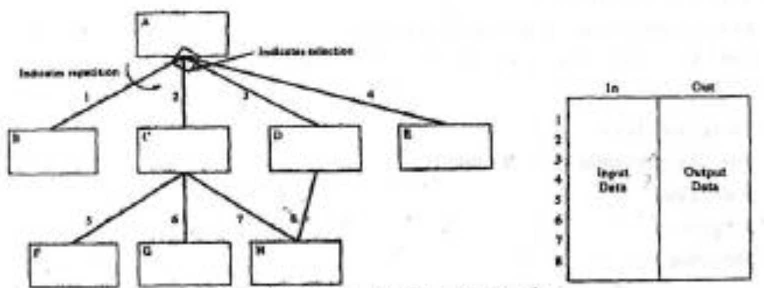


Figure 4.5 Format of a structure chart.

system. A structure chart from a flow chart in two ways: a structure chart has no decision boxes and the sequential ordering of tasks inherent in a flow chart can be suppressed in a structure chart.

The structure of hierarchical system can be specified using a structure chart s is the following figure.

4.3.3. HIPO Diagrams

HIPO diagrams (Hierarchy-Process-Input-Output) were developed at IBM as design representation schemes for top-down software development and as external documentation aids for released products.

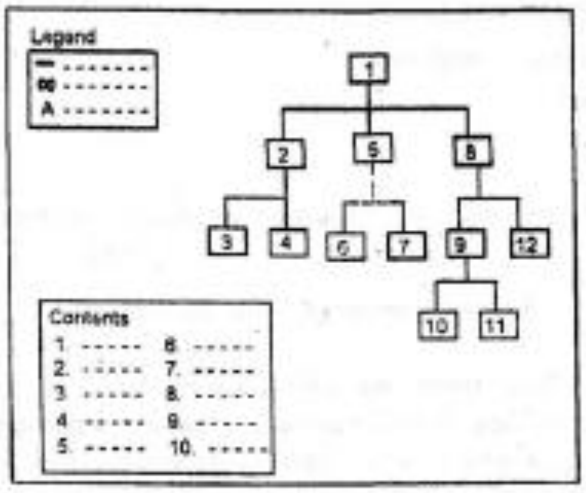


Fig. 4.6 HIPO diagram

A set of HIPO diagrams contain a visual table of contents, a set of overview diagrams and a set of detail diagrams. The visual table of contents is a directory to the set of diagrams in the package; it consists of a tree-strucured directory, a summary of the contents of each overview diagram and a legend of symbol definitions. The visual table of contents is a stylized structure chart.

Overview diagrams specify the functional processes in a system. Each over view diagram describes, the input, processing steps, and output for the function being specified.

4.3.4. Procedure template

In the early stages of architechtural design, only the information in level 1 need be supplied. As design progresses, the information on levels 2,3 and 4 can be included in successive steps.

Procedure Name
Part of (subsystem name & number)
Called by: LEVEL 1
Purpose:
Designer/Date(s)
Parameters: (names, modes, attributes, purposes)
Input Assertion: (Preconditions) LEVEL 2
Output Assertion: (Post conditions)
Globals: (names, modes, attrributes, purposes shared with)
Side effects:
Local Data Structures: (names, attributes, purposes)
Exceptions: (conditions, responses) LEVEL 3
Timing Constraints:
Other Limitations
Procedure Body: (pseudo code, Structured English, Structured flow chart, decision table) LEVEL 4

Fig. 4.7 Format of procedure template.

The term "Side effect" means any effect a procedure can exert on the processing environment that is not evident from the procedure name and parameters. Modifications to global cariable, reading or writing a file, opening or closing a file or calling a procedure that inturn exhibits side effects are all examples of side effects.

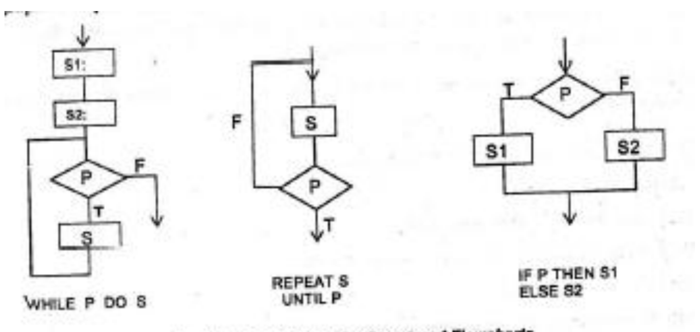


Fig 4:9 – Basic Forms for Structured Flowcharts

It is recommended that only information on level be provided during initial architechtural design because detailed specification of side effects, exception handling, processing algorithms and concrete data representations will sidetrack the designer into appropriate levels of detail too soon,
During detailed design, the processing algorithms and data structures can be specified, using structured flowcharts, pseudocode or Structured English.
Procedure interface specifications are effective notation of architechtural design when used in combination with structure charts and data flow diagrams. They also provide a natural transaction from architechtural to detailed design and from detailed design to impelementation.

4.3.5. Pseudo code

Pseudo code notation can be used in both the architechtural and detailed level of abstraction using pseudo code, the designer describes system characteristics using short. concise, English language phrases that are structural by key words such as If-then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Using the top-down design strategy, each English phrase is expanded into more detailed pseudo code until the design specification reaches the level of detail of the implementation language.

Pseudo code can replace flowcharts and reduce the amount of external documentation required to describe a system

```
INITIALIZE tables and counters, OPEN files
READ the first text record
WHILE there are more text records DO
WHILE there are more words in the text record DO
EXTRACT the next record
SEARCH word table for the extracted word
—
IF the extracted word is found THEN
INCREMENT the extracted word's occurence count
ELSE .
INSERT the extracted word into word table
ENDIF
INCREMENT the words processed counter

END WHILE at the end of the text record
END WHILE when all text records have been processed
PRINT the word_table and the words_processed counter
CLOSE files
TERMINATE the program
```

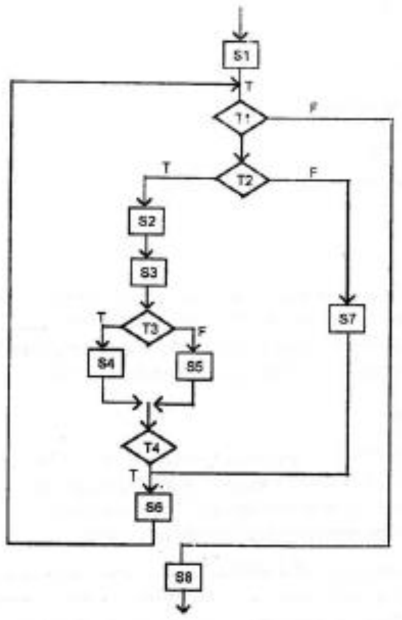
Fig. 4.8 An example ofapseudo code disign specification

4.3.6. Strucutured flowcharts

Flow charts are the traditional means for specifying and documenting algorthmic details in a software systems. Flowcharts incorporate rectangular boxes for actions, diamond shaped boxes for decisions, directed arcs for specifying interconnection between boxes and a variety of specially shaped symbols to denote input,output, data stores etc.

Structured flow charts differ from traditional flowcharts in that structured flowcharts are restricted to compositions of certain basic forms. This makes the resulting flowchart the graphical equivalent of a structured pseudocode description.

The basic forms are characterized by single entry into and single exit from the form. Thus, forms can be nested within forms to any arbitrary depth and in any arbitrary fashion, so long as the single entry, single exit property is preserved. A composite structured flowchart and the pseudo code equivalent are illustrated in the following example.



```
S1
WHILE T1 DO
IF T2 THEN
S2
REPEAT
S3
IF T3 THEN
S4
ELSE
S5
END IF
UNTIL T4
ELSE
S7
END IF
S6
END WHILE
S8
```

Fig. A structured flowchart and pseudocode equivalent.

Because structured flowcharts are logically equivalent to pseudo code, they have the save expressive power as pseudocode; both can be used to express any conceivable algorithm. However structured flowcharts may be preferred in situations where clarity of control flow is to be emphasized.

The single entry, single exit property allows hierarchical nesting of struptured flowchart constructs to document a design in top-down fashion, starting with top level structure-ancLproceeding through detailed design.
Structured flowcharts make it easy to violate the single entry, single exit property and they are not machinereadable or machinemodifiable as is pseudocode.

4.3.7. Structured English

Structured English can be used to provide a step by step specification for an algorithm. Like pseudocode, Structured English can be used at any desired level of detail. Structured English is often used to specify coobook recepies.

1. Preheat oven to 350 degrees F
2. Mix eggs, milk and vanilla
3. Add flour and baking soda
4. Pour into a greased baking dish
5. Cook until done.

4.3.8. Decision Tables

Decision tables can be used to specify complex decision logic in a high-level software specification. They are also useful for specifying algorthmic logic during detailed design. At the level of usage decision tables can be specified and translated into source code logic. Several preprocessor packages are available to translate decision tables into COBOL.

Design Techniques

The design process involves developing a conceptual view of the system, establishing system structure identifying data streams and data stores, decomposing high level functions into subfunctions, establishing relationships and interconnection among componenets. developing concrete data representations and specifying algorthmic details.

Developing a conceptual view of a software system involves determining the type of system to be built. The system may be a data-base system, a graphics system, a telecommunication system, a process control system or a data process system; or the system may combine aspects of different system types. In each of these application areas there are certain viewpoints, terminology, tools and notations suitable to that class of applications. It is essential that the software design team have a strong conceptual understanding of the nature of the system to be constructed and be familiar with the tools and techniques in the appropriate application areas. It is not uncommon for a design team to be composed of one or more specialists from each appropriate area.

A data store is a conceptual data structure. During external and architechtural design, one may identify the need for a stack, queue, or file for example, but the exact implementation details of the data structures should be deffered until detailed design. Detailed design decisions should be delayed as long as possible. Early binding of design decisions particularly in regard to representation details for data structures can result in a system structure that is difficult to modify during subsequent development and maintanence activities. During external and architechtural design, data structures should be defined as data abstractions, with emphasise placed on the desired operations and not an implementation details.

Data streams and data stores can be specified using data flow diagrams, data dictionaries and data' abstraction techniques. Decomposition of high level functions can be initiated from data flow diagrams and often involves use of structure diagrams, HIPO diagrams and procedures specifications.

During the past few years. Several techniques have been developed for software design. These techniques include stepwise refinement, levels of abstraction, Structured design, integrated top-down development and the Jackson design method. Although these techniques are often called "design methodologies" they are infact viewpoints and guidelines for the design process.

Design techniques are typically based on the "top-down" and/ or "bottom-up" design strategies. Using the top-down approach, attention is forst focused on global aspects of the overall system. As the design progressed, the sytem is decomposed into subsystems and more consideration is given to specific issues. Backtracking is fundamental to top-down design. Inorder to minimize backtracking, many designers advocate a mixed strategy that is prodominately top-down but involves specifying the lowest-level modules first. The primary advantage of the top-down strategy is that attention is first directed to the customers needs, user interfaces and the overall nature of hte problem being solved.

In the bottom-up approach to software design, the designer first attempts to identify a set of primitive objects, actions and relationships that will provide a basis for problem solution. Higher-level concepts are their formulated in terms of the primitives. The bottom-up strategy requires the designer to combine features provided by the implementation language into more sophisticated entities. These entities are combined inturn until a set of functions, data structures and inter connections has been constructed to solve the problem using elements available in the actual programming environment. Bottom-up design may also require redesign and design backtracking.

The success of bottom-up design depends on identifying the "proper" set of promitive ideas sufficient to implement the system.

Several design techniques are described in the following sections.

4.4.1. Stepwise Refinement

Stepwise refinement is a top-down technique for decomposing a system from high-level specificatiors into more elementary levels. Stepwise refinement is also known as

"step wise program development" and "successive refinement". It involves the following activities.
1) Decomposing design decisions to elementary levels.
2) Isolating design aspects that are not truly interdependent.
3) Postponing decisions concerning representation details as long as possible.
4) Carefully demonstrating that each successive step in the refinement process is a faithful expansion of previous steps.

Incremental addition of detail at each step in the refinement process postpones design decisions as long as possible and allows the designer to argue convincingly that the resulting software product is consistent with the design specifications.

Stepwise requirements begins with the specification derived during requirements analysis and external design. The problem is first decomposed into a few major processing steps that will demonstrably solve the proble, The process is then repeated for each part of the systems until it is decomposed in sufficient detail so that implementation in a executable programming language is straight forward.

Successive refinement can be used to perform detailed design of the individual modules in a software product. The following example, illustrates detailed design by the method of successive refinement. A routine is required to write the first N prome numbers into file F, N is a formal parameter in the routine and F is globally known,

The initial version of the routine might be:

Initial version :

```
Procedure PRIME (N: integer);
Var I,X: integer;
begin
rewrite (F);
X:=l;
For 1=1 to N do <-
begin
X: = "next prime number";
write (F,X)
end;
end {PRIME};
```

The next step is to refine the statement X: = "next prime number". By introducing a Brolean Variable, PRIM, the statement can be expressed as

Refinement I:

```
repeat
        X:=X+1;
        PRIM :="X is a prime number"
until PRIM;
```

All prime number except the first (which is 2) are odd. We can treat 2 as a special case and increment X by 2 in the repeat loop. Refinement 2:begin rewrite(F)' X:=l;

```
ifN>=l then
        write(F,2);
For I:=2 to N do
begin
        repeat X:=X+2;
PRIM :="X is a prime number"
until PRIM
Write (F,X);
end
end
```

The next refinement invloves determining whether X is a prime number. X is prime if it is divisible only by 1 and itself with Zero remainder.

Refinement 3

```
repeat
        X = X2
        Kl = 2
        repeat
                K = K+1
                PRIM = "N is not divisible by K"
        until PRIM
until not(PRIM) or (K=X-1)
```

The remainder function med can be used to determine whether X is divisible by K.

Refinement 4

```
PRIM = (X mod X) =0
```

out procedure now has the following structure.

```
Procedure PRIME (N integer)
Var I,X,K integer
PRIM boolean
begin
rewrite(F), X=l;
if N>=1 then
        write (F,2)
Fori := 2 to N do
begin
        repeat
                X = X+2, K:=2
                repeat
                        K = K+l
                        PRIM = (X mod K) = 0
                        until not (PRIM) or (K=X-1)
                until PRIM
                write (F,X)
        end
end.
```

The procedure for computing the first N prime numbers and writing them in file F is complete. It is complete because each pseudocode statement has been refined into executable source code. However the efficiency of the program can be vastly improved in several ways. For example, it is not necessary to check X for divisibility by every number K between S and X-l. Recall that very non prime number can be expressed as the product of prime factors. If X is divisible by a nonprime number K, it is also divisible by the prime factor of K. Thus, we can keep the prime numbers already calculated in an array P and only check X for divisibility by the elements of P. This results in

Refinement 5

```
X=l, P[l] = 2, LIM = 1.
For I = 2 to N do begin
repeat
X = X+2; K = 2, PRIM := true;
while PRIM and (K< LIM) do
begin
        PRIM := (Xmod P[k]) <> 0
        K:=K+1; .
end;
```

```
until PRIM;
P[l] := X:LIM := LIM + 1
end;
```

The major benefits of stepwise refinement as a design technique are

1. top-down decomposition
2. Incremental addition of detail
3. Postponement of design decisions
4. Continual Verification of Consistency

4.4.2. Levels of abstraction

Levels of abstraction was originally described by Dijkstra as a bottom-up design technique in which an operating system was designed as a layering of hierarchical levels starting at level o(process or allocation, real-time clock interrupts) and building up to the level ol processing independent user programs. In Dijkstra's system each level of abstraction is composed of a group of related functions, some of which are externally visible and some of which are externally visible and some of which are internal to the level.

Internal functions are hidden from oihci levels: ihey can only be invoked by functions on the same level. The internal functions arc used to perform tasks common to the work being performed on that level of abstraction. Of cource, functions on higher levels cannot be used functions on lowest levels; function usage establishes the level of abstraction.

Each level of abstraction performs a set of services for the functions on the next higher level of abstraction. Thus a file maflipulatBon system might be layered as a set of routines to manipulate fields (bit vector on level 0) a set of routines to manipulate records (set of fields on level 1) and a set of routines to manipulate files (set of records on level 2).

Each level of abstraction has exclusive use of certain resources (I/O devices, data structures) that other levels are not permitted to access.

Higher level functions can invoke functions on lowest levels, but lower-level functions cannot invoke or n any way make use of higher-level functions. This later restriction is important because lower levels are then self-sufficient for supporting other abstractions. The lower-levels can be used without change as the lower-level routines in other applications or in adaptations and modifications to an existing systems. The level of abstraction utilized in the T.H.E. Operating system are listed in

Level 0	Processor allocation clock interrupt handling
Level 1	Memory segment controller
Level 2	Console messageinterpreter
Level 3	I/O buffering
Level 4	User programs
Level 5	operator

Table : Level of abstraction in the T.H.E. operating system

4.4.3. Structured design

Structured design was developed by constantine as a top-down technique for architectural design of software systems. The basic approach is structured design is systematic conversion of data-flow diagrams into structure charts. Design heuristics such as coupling and cohesion are used to guide the design process. As discussed earlier coupling measures the degree to which two distinct modules are bound together and cohesion is a measure of the relationship of elements within a module to one another. A well designed systems exhibits a low degree of coupling between modules and a high degree of cohesion among elements in each module.

The first step in structured design is review and refinement of the data flow diagram(s) developed during requirements definition and external design. The second step is to determine whether the system is transform-centered or transaction driven and to derive a high level structure chart based on this determination. In a transform-centered system the data flow diagram contains Input, Processing and Output segments that are converted into Input. Processing and Output subsystems is the structured chart Boundaries between the three major subsystems in a transform-centered system are identified by determining the point of most abstract input data and the point of most abstract output data.on the data flow diagram. The situation is illustrated in the following figure.

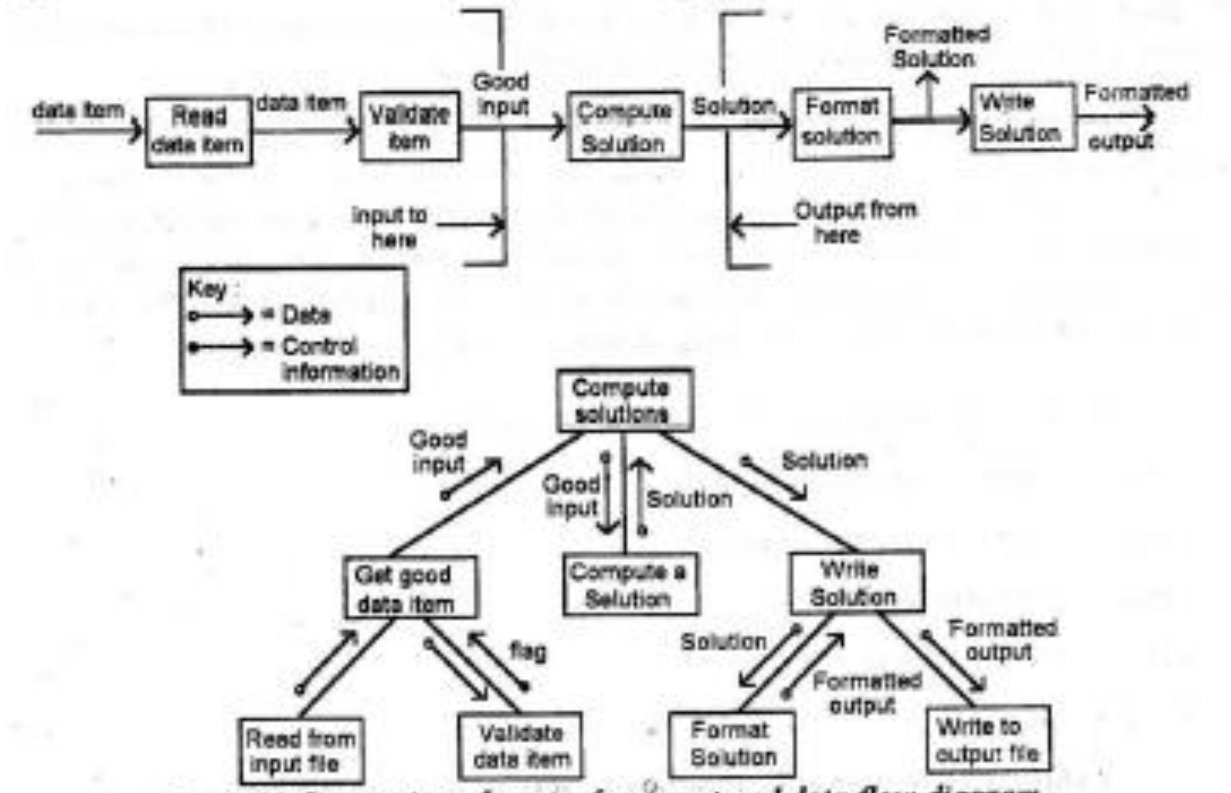


Fig. 4.11 Conversion of a transform-centered data flow diagram into an input, process, output structure chart.

The point of most abstract input data is the point in the data flow diagram where the input stream can no longer be identified. Similarly, the point of most abstract output data is the point in the data flow diagram where components of the output data stream can first be identified.

The third step in structured design is decomposition of each subsystem using guidelines such as coupling, cohesion, information hiding, levels of abstraction, data abstraction and the other decomposition criteria discussed earlier.

According to constantine a hierarchical tree structure is the solution from that usually results in the lowest cost implementation.

Decomposition of processing functions into modules should be continued until each module contains no subset of element that can be used alone and until each module is small enough that its entire implementation can be grasped at once.

Data dictionary can be used^ in coxy unction with a structure chart to specify data attributes relationships among data items and data sharing among modules in the system.

In addition to coupling, cohesion, data abstraction information hiding and other decomposition criteria the concepts of "Scope of Effect" and "Scope of control" can be used to determine the relative positions of modules in a hierarchical framework. The "Scope of control" of.a module is that module plus all modules that are subordinate to it in the structured chart. The scope of control of module B is B, D and E in the following chart.

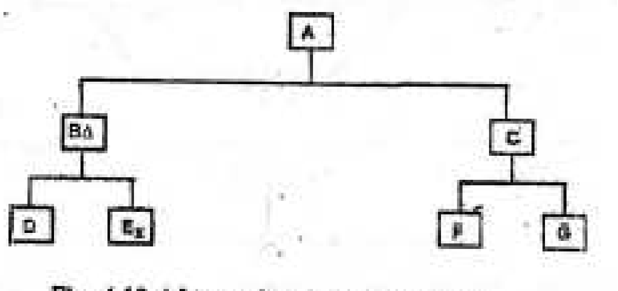


Fig. 4.12 A hierarchical structure chart

The "Scope of effect" of a decision is the set of all modules that contain code that is xecuted based on the outcome of that decision. In general, systems are more loosely coupled when the scope of effect of a decision is within the scope of control of the module containing the decision. The following example illustrates the situation.

In the above diagram, suppose execution of some code in module B depends on the outcome of a decision, ?£, in module E.

Either E will return a control flag to B or the decision process will have to be repeated in B. The former approach requires additional code to implement the flag and results is control coupling between B and E. The later approach requires duplication of some of E's code in module B. Duplication of code is inefficient and causes difficulties in co-ordinating changes to both copies if the decision process should change. The situation can be remedied by modifying the system so that the scope of effect of decision X is within its scope of control. Moving the decision process upward into B or moving the code in B effected by the decision into E will produce the desired effect.

Detailed design of the individual module in a system can be accomplished using the succuessive refinement techniques and by detailed design notations such as HIPO diagram, procedure templates, pseudocode and /or Structured English.

A transform-centered system is characterized by similar processing steps for each data items processed by the Input, Process and Output subsystems. In a transaction-driver system one of several possible paths through the data flow diagrams is traversed by each transaction. The path traversed is typically determined by user input commands.

Transaction-driver systems have data-flow diagrams of the form illustrated follows,

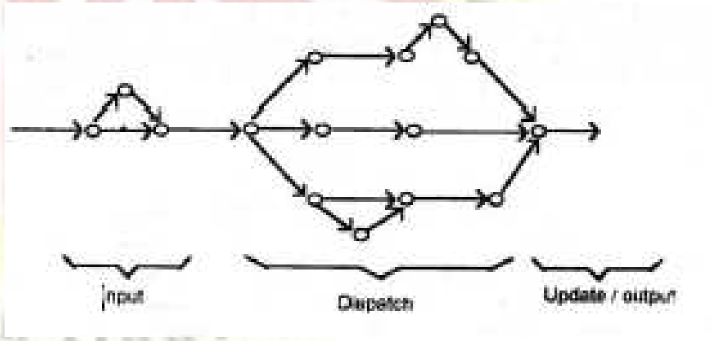


Fig. 4.13 Transact ion-driver data flow diagram.

This figure can be converted into a structure chart having Input, controller, Dispathcher and Updata/output subsystems as illustrated as following figure.

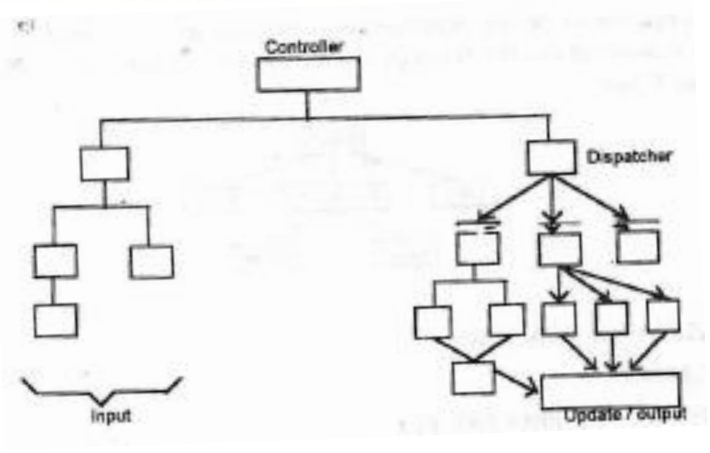


Fig. 4.14 Transaction driven Structure chart.

In summary, the primary benefit of structured design arethe use of data flow diagrams focuses attention on the problem structure. This follows naturally from requirements analysis and external design.

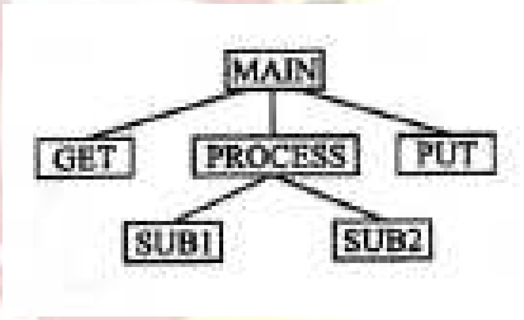
The method of translating data flow diagram into structure chart provides a method for initiating architechtural design in a systematic manner.

3. Data dictionaries can be used in conjunction with structure charts to specify data attributes and data relationships.
4. During heuristics such as coupling and cohesion, and scope of effect and scope of control provide criteria/or systematic development of architechtural structure and for comparison of alternafice design structures.
5. Detailed design techniques and notations such as successive refinement, HIPO diagrams, procedure specification forms, and pseudo code can be used to perform detailed design of the individual modules.

4.4.4. Integrated Top-down development

Integrated top-down development integrates desrgn, implementation and testing. UsiAg^ integrated top-down development, design proceeds top-down from the highest-level routines; they have the primary function of co-ordinating and sequencing the lower-level routines. Lower-level routines may be implementations of elementary functions or they mayjintur'ninwke rrwe primitive routines. There is thus a hierarchical structure to a top-down system in vy/hich routines can invoke level routines but cannot invoke routines on a higher-level.

The integration of design, implementation and testing is illustrated by the following example. It is assumed that the design of a system has proceeded to the point illustrated in the following figure.



STRATEGY: DESIGN MAIN CODE MAIN STUBS FOR GET, PROCESS, PUT TEST MAIN .DESIGN GET CODE GET TESTMAIN GET DESIGN PROCESS CODE PROCESS STUBS FOR SUB1, SUB2 TEST MAIN GET, PROCESS

DESIGN PUT
CODE PUT
TEST MAIN, GET, PROCESS, PUT
DESIGN SUB1
CODE SUB1
TEST MAIN, GET, PROCESS, PUT, SUB1
DESIGN SUB2
CODE SUB2
TEST MAIN, GET, PROCESS, PUT, SUB1, SUB2.

Fig. 4.15 e.g. Integrated top-down development strategy.

The purpose of procedure MAIN is to co-ordinate and sequence the GET, PROCESS, and PUT routines. These three routines can communicate only through^MAIN, Similarly, SUB1 and SUB2, can communicate only through PROCESS.

The implementation and testing strategy for the example might be a illustrated in above figure. The stubs referred are dummy routines written to simulate subfunctions that are invoked higher-level functions. As coding and testing progresses, the stubs are expanded into full functional units that may inturn require lower-level stubs to support them.

Stubs can fulfil a number of useful purposes prior to expansion into full functionality. They can provide output messages, test input parameters, deliver simulated output parameters and simulate timing requirements and resource utilization. Use of program stubs in top-down development provides an operational prototype of hte system as development progresses.

Some designers restrict data communication between modules to the parameter lists, while other designers allow global variables that are common to two or more modules. A reasonable compromise is to communicate data between levels via parameter lists and to permit access to common global data by modules in the same level of hierarchy. This approach is particularly attractive when each hierarchical level is identified as a level of abstraction in the system. Each level of abstraction provides supporting functions for the next higher level in the hierarchy and is supported by the_levels below it.

The integrated top-down design provides an orderly and systematic framework for software development.

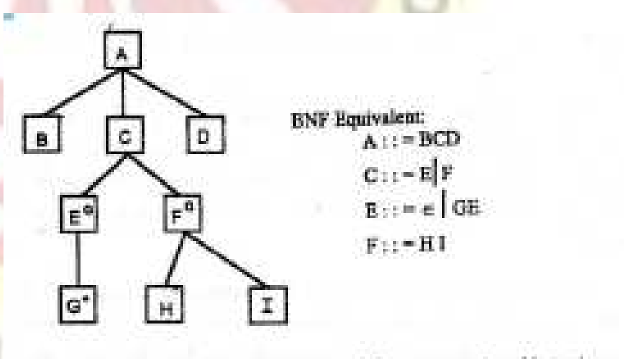
The primary diadvantage of the integrated top-down approach is that early high-level **design decisions may have to be reconsidered when the design progresses to lower** levels. This **may require** design **backtracking and considerable rewriting of code. These are other disadvantages to integrated top-down development, the system may be a** very expensive test **harness for** newly **added** procedures; it **may** not **be possible to** find high-level test data to exercise newly added procedures in the desired manner; **and** in certain instances such asinterrupt handlers and I/o drivers, procedure stub may not be suitable.

4.4.5. Jackson structured programming

Jackson Structured Programming was developed by Michael Jackson as a systematic technique for mapping the structure of a problem into a program structure to solve the problem. The mapping is accomplished in three steps.

1. The problem is modeled by specifying the input and output data structures using tree structures diagrams.
2. The input-output model is converted into a structural model that contains the operations needed to solve the problem.

Input and output structures are specified using a graphical notation to specify data hierarchy, sequence of data, repitition of data items and alternate data item. Specification of data item A is illustrated as



f object A using Jackson Structured Programming Notation

According to this notation, item A consists of a B followed by a C followed by a D (reading left to right on the same level) B and D have no substructures. C consists of either an E or an F (denoted by "o"). E consists of Zero or more occurances of G (denoted by "*"), and F consists of an H followed by an I. This notation is the graphical equivalent of regular expressions.

The second step of the jackson method involves converting the input, and output structures into a structural model of the program. This is accomplished by identifying points of commonilty in the input and output structures and combining the two structures into a program structure that maps inputs into outputs. Labels on data items in the resulting structure are converted to process names that perform the required processing of the data items.

The third step expands the structural model of the program into a detailed design model containing the operations needed to solve the problem. This step is performed in three substeps.

1. A list of operations required to perform the processing steps is developed.
2. The opreations are associated with the program structure.

3. Program Structure and operations are expressed in a notation called Schematic logic, which is stylished pseudocode. Control flow for selection and iteration are specified in this step. The following example illustrates the basic concepts of the Jackson method. An input file consists of a collection of inventory records sorted by part number. Each record contains a part number and the number of units of that item issued or received in one transaction. An output report is to be produced that contains a heading and a net movement line for each part number in the input file. Because the input file is sorted by part number, all issues and receipts of a given part number are in a continuous portion of the file called a part group. Each record is a part group is called a movement record.

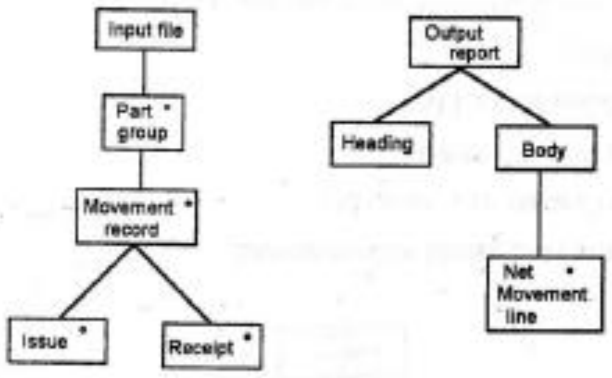
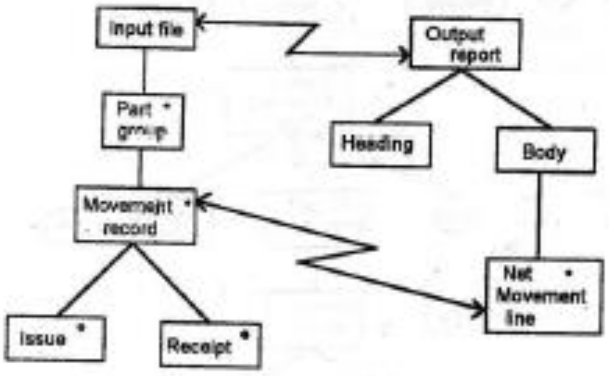


Fig. 4.17 Input and output structures for an inverting problem

respondence between input and output structures are illustrated as



The input and output strucutres are illustrated.

Fig. 4.1 u Correspondence between input & output structure for an inventory problem

The program structure is derived by superimposing the input file structure on the output report structure and overlaying the corresponding nodes in the two graphs. The resulting program structure, and anslating the annotated structure diagram into schematic logic (pseudocode). These steps are illustrated the following figures.

1. OPEN FILES
2. CLOSE FILES
3. STOP RUN
4. READ A RECORD INTO PART-NUM, MOVMNT
5. WRITE HEADING
6. WRITE NET-MOVEMENT LINE
7. SET NET-MOVMNT TO ZERO
8. ADD MOVMNT FROM NET-MOVMNT
9. SUBTRACT MOVMNT FROM NET-MOVMNT

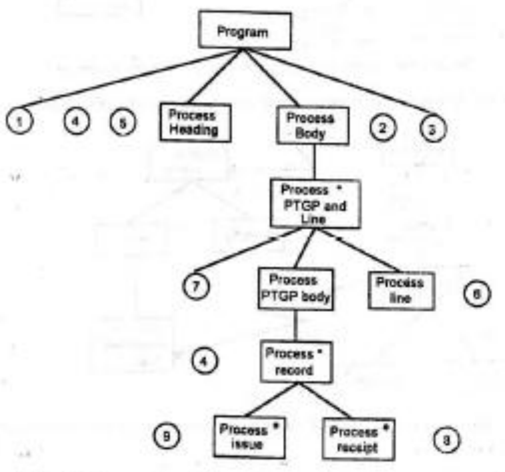


Fig. 4.19 Association of operations with program structure.

```
BEGIN PROGRAM
OPEN FILES
READ PART-NUM, MOVMNT
WRITE HEADING
ITERATE WHILE NOT(END-OF-FILE)
```

```
SET NET-MOVMNT TO ZERO
ITERATE WHILE SAME-PART-NUMBER
IF(MOVMNT=ISSUE) THEN
SUBTRACT MOVMNT FROM NET-MOVMNT
ELSEIF (MOVMNT = RECEIPT) THEN
ADD MOVMNT TO NET-MOVMNT
ENDIF
READ PART-NUM. MOVMNT
ENDWHILE
WRITE NET-MOVEMENT LINE
END WHILE
CLOSE FILES
STOP
END PROGRAM
```

Fig. 4.20 Schematic logic representation of an inventory program.

Difficulties encountered in applying the Jackson method include ^strucuture clashes and the need for look-ahead. Structure plashes occur when points of commonality between input and output data structures cannot be identified structure clashes can be resolved by a technique called program inversion. Using program inversion, a consumer routine calls on a producer routine to deliver the next required data item.

More recently, Micheal Jackson has developed a method for software design called the Jackson Design method. This approach involves modeling the real world phenomenon of interest as a network of sequential processes that communicate using serial data streams. Detailed design of the processes can be accomplished using Jackson structured programming as described above.

Jackson structured programming is widely used and is quite effective in situations where input and output data structures can be defined in a precise manner. It appears to be most effective in data processing applications.

Detailed Design Consideration

Detailed design is concerned with specifying algorthmic details, concrete data representations, interconnections among functions and data structures and packaging of the software product. Detailed design is strongly influenced by the implementation language, but it is not the same as implementation, detailed design is more concerned with systematic issues and less concerned with systematic details than is implementation.

The starting point for detailed design is an architectural structure for which algorithmic details and concrete data representations are to be provided. While there is a strong temptations to proceed directly from architechtural structure to implementations, there are a number of advantages to be gained in the intermediate level of detail provided by detailed design.

Detailed design seperates the activity of low-level design from implementation, just as the analysis and design activities isolate consideration of what is described from the structure that will achieve the describe result. An adequate detailed design specification minimizes the number of surprises during product implementation.

Detailed design activities inevitably expose flaws in the architechtural structure, and the ensuring modifications will be eased by having, fewer details to manipulate than would be present in the implementation language. Detailed design also provides a vehicle for design inspections, structured walkthroughs and the critical Design review. Notations for detailed design include HIPO diagrams pseudo code, Structured English, Structured Flowcharts, data structured diagrams and physical layouts for data representations. The detailed design representation may utilize key words from the implementation language to specify data representation.

Real time and distributed system design

Many of the popular design "methodologies" were developed-as_design techniques for applications programs, operating systems and utility programs. These methods support concepts such as hierarchical decomposition, modularity, information hiding and data abstractions. These design concepts are important, for real-time and distributed systems, but they donot explicitly incorporate methods to deal with performance characteristics or problems of network configuration and communication protocols.

According to Franta, a distributed system consists of a collection of nearly autonomous processors that communicate to achieve a coherent computing system. Each processor possesses a private memory, and processors communicate through an interconnection network.

Major issues to be addressed in designing a distributed system include specifying the topology of the communication network, establishing rules for accessing the shared communication channel allocating application processing functions to processing functions to processing nodes in the network and establishing rules for process communication and synchronization. The design of distributed systems is further complicated by the need to allocate network functionality between hardware and software cmponents of the network.

By definition, real-time systems must provide specified amounts of computation with in fixed time intervals. Real-time systems typically sense and control external devices, respond to external events, and share processing time between multiple tasks processing demands are both cyclic and event-driven in nature. Event-driven activities may occur in bursts, thus requiring a high ratio of peak to average processing. Real-time systems often form distributed networks, local processors may be associated with sencing devices and actuators.

A real-time network for process control may consist of several minicomputers and micro computers connected to one or more large processors. Each small processor may be connected to a cluster of real-time devices. In this manner, processing power can be placed at "natural" sites in the system, and data can be processed at the point of recption rather than at a central node, thus reducingjcommunrcation bandwidth needs. Decomposition criteria for distributed real-time systems include the need to maintain process simplicity and to minimize interprocess communication bandwidths by communicating simple processed messages rather than raw data.

Process control systems often utilize communication networks having fixed, static. topology and known capacity requirements. In contrast, more elaborate real time systems provide dynamic reconfiguration of the network topology and support unpreditable load
demands.

Many process control systems are designed with only two levels of abstraction, which comprise basic system functions and application programs. It is not clear whether a two level hierarchy is the result of process control system characteristics or process control designer characteristics.

In summary, the traditional considerations hierarchy, information hiding and modularity are important concepts for the design of real-time systems. However these concepts are typically applied to the individual components of a real-time systems. Higher-level issues of networking, performance and reliability must be analyzed and designed before the component nodes or processes are developed.

Test plan

The test plan is an important, but often overlooked, product of software design. A test plan prescribes various kinds of activities that will be performed to demonstrate that the software product meets its requirements.

The testplan specifies the objecitives of testing, (e.g. to acheive error-free operation under stated conditions for a stated periods of time), the test completion criteria (to achieve a specified rate of error exposure, to achieve a specified percent of logical path coverage), the system integration plan (strategy, Schedule, responsible individuals), mehtods to be used on particular modules (walkthroughs, inspections, static analysis, dynamic tests, formal verification) and the particular method to be used on particular molecules walk through, inspections, statio analysis, dynamic belts format verification and the particular test cases to be used.

There are 4 types of tests that a software product must satisfy functional tests, performances tests, and structural tests. Functional tests and performance tests are based on the requirements specifications, they are designed to demonstrate that the system satisfies its requirements. Therefore, the test plan can be only as good as the requirements, which in turn must be phrased in quantified, testable terms.

Functional test cases specify typical operating conditions, typical input values, and typical expected results. Functional tests should also be designed to test boundary conditions just inside and just beyond the boundaries, (e.g. square root of negative numbers, inversion of one by one matrices, etc.) Also, special values such as files and arrays containing identical values, the identity matrix, the zero, matrix etc. should be tested. Assumed initial values and system defaults should be tested and inputs having assumed ordering relations should be tested. Assumed initial values and system defaults should be tested and inputs having assumed ordering relations should be tested with data that have both correct and incorrect ordering.

Performance tests should be designed to verify response time (under various load) execution time throughput, primary and secondary memory utilization and traffic rates on data channel and communication links. Performance list will often indicate processing bottlenecks to be addressed during system testing and tuning.

Each functional list and performance test should specify the machine configuration assumptions concerning the system status for the test case, the requirements being tested the test inputs and the expected results. It is particularly important that the expected results of each list be specified prior to system implementation and actual testing. Otherwise, it is eas to rationalize an incorrect result.

Stress tests are designed to overload a system in various ways, such as attempt to sign on more than the maximum allowed number of terminals, processing more than the allowed number of identifiers or static levels or disconnecting a communication link. The purpose of stress testing are to determine the limitations of the system and , when the system fails, to determine the manner in which the failure is manifest. Stress tests can provide valuable insight concerning the strengths and weakness of a system. Stress lists are derived from the requirements, the design and the hunches and intuitions of the designers. Structural lists are concerned with examining the internal processing logic if a software system. The particular routines called and the logical paths traversed through the routines are the objects of interest. The goal of structural testing is to traverse a specified number of paths through in the system to establish thoroughness of testing.

The recommended approach is to perform the functional performance and stress tests on the implement system and to augment these lists with additional structure tests to achieve the desired level of test coverage. Thus, structural tests cannot be designed until the system is. implemented and subjected to the predefined test plan.

Distinguish between Black Box and White Box Testing

Black box requires no knowledge about the minute internal working of the module concerned. This test is conducted to check the total functionality of the module. If the module is not producing results as expected the error has to be identified and rectified. This type of testing is not done normally for modules. It is applied on the product as a whole,

White box testing requires complete knowledge of the individual modules, all its independent paths, all logical and decision making checks and also its test case design. This type of test is an exhaustive test. Such exhaustive test are necessary because of the varied nature of software errors, logical errors, path errors, typo graphical errors, syntax errors and validation errors.

Black Box	**White Box**
1. It is a functional testing	1. It is an individual path testing
2. Overall knowledge about the software is enough	2. To conduct white box testing a through knowledge of the internal arrangements, control structures are essential
3. It can be conducted by any body.	It can be conducted by those having complete idea (or) by the developerhimself.
4. The logical structure and the internal logical structure of the software.	Black box test does not many about the procedure are the main concern.
5. Test cases are generated to check individual program (or) software as a whole.■	Test cases are generated to verify the paths only.
6. Black box test is very simple.	6. It is very difficult complex. Sometimes exhaustive testing are reduced to selective testing.
It cannot be used to check inface errors.	7. It can be used to validate interfaces.

Mile stones walk through and inspection.

One of the most important aspects of a systematic approach to software development is the resulting visibility of the evolving product. The system becomes- explicit, tangible and assessable products of analysis and design to be examined during system development include specifications for the externally observable characteristics of the system, the evolving user's manual architectural design specifications detailed design specifications and the test plan.

Development of these intermediate work products the opportunity to establish milestones and to conduct inspections and reviews. These activities in turn expose errors, provide increased project communication, keep the project on schedule and permit verification that the design satisfies the requirements.

The two major milestones during design are the Preliminary Design Review (PDR) and the Critical Design Review (CDR). The PDR is typically held near the end of architectural design and prior to detailed design CDR occurs at the end of detailed and prior to implementation.

Depending on the size and complexity of the product being developed, the PDR and CDR may be large formal affairs involving several people are they may consists of on informal meeting between a programmer and they project leader. In any case, a formal sign-off should occur to indicate that the milestone has been achieved.

The major goal of the PDR is to demonstrate that the externally observable characteristics and architectural structure of the product will satisfy the customer's requirements. Functional characteristics, performance attributes, external interfaces, user dialogues, report formats, exception conditions and exception handling, product subsets and future enhancements to the product should all the reviewed during that PDR.

One of the outcomes is likely for a PDR, the PDR may expose enough serious problems that a subsequent PDR is scheduled or the PDR may expose only minor errors that can be corrected and reviewed at CDR.

The CDR is held at the end of detailed design and prior to implemetation. Among other things, CDR provides a final management decision point to build or cancel the system, The CDT is in essence a repeat of the PDR, but with the benefit of additional design effort.

The involvement of customers in design reviews is a sensitive and difficult issue. The product designers may benefit from increased customer involvement on the other hand, the customer may use the design reviews to interject changes to the requirements that necessitate significant design changes.

4.7.1. Walkthroughs and Inspections

A structured walkthrough is an indepth technical revew of some aspect of a software system. Walkthrough can be used at any time, during any phase of a software project. Thus, all or any part of the software requirements, the architechtural design specifications, the detailed design specification can be reviewed at any stage of evolution

A walkthrough team-consists of four to six people. The person whose material is being reviewed is responsible for providing copies of the review material to members of the walkthrough team in advance of the walkthrough session, and team members are responsible for reviewing the material prior to the session. During the .walkthrough the receiver "walks through" the material prior to the errors, request clarifications and explore problem area in the nwaterial under review.

It is an important to emphasize that the material is reviewed, and not the reviewee, the focus of a walkthrough is on detection of errors and not on corrective actions.

Design inspections are conducted by teams of trained inspectors who work from checklists of items to examine. Special forms are use^To~record-problems encountered. A typical design inspection team consists of a moderator / sectretary a designer, an implementor, and a tester. The designer, implementor and tester may or may not be the people responsible for actual design, implementation and testing of the product being inspected.

Unit-4:

Implementation issues :

Code-reuse - Programming interfaces of present-day languages are very sophisticated and are equipped huge library functions. Still, to bring the cost down of end product, the organization management prefers to re-use the code, which was created earlier for some other software. There are huge issues faced by programmers for compatibility checks and deciding how much code to re-use.

Version Management - Every time a new software is issued to the customer, developers have to maintain version and configuration related documentation. This documentation needs to be highly accurate and available on time.

Target-Host - The software program, which is being developed in the organization, needs to be designed for host machines at the customers end. But at times, it is impossible to design a software that works on the target machines.

Structured Coding techniques

In the process of coding, the lines of code keep multiplying, thus, size of the software increases. Gradually, it becomes next to impossible to remember the flow of program. If one forgets how software and its underlying programs, files, procedures are constructed it then becomes very difficult to share, debug and modify the program. The solution to this is structured programming. It encourages the developer to use subroutines and loops instead of using simple jumps in the code, thereby bringing clarity in the code and improving its efficiency Structured programming also helps programmer to reduce coding time and organize code properly.

Structured programming states how the program shall be coded. Structured programming uses three main concepts:

Top-down analysis - A software is always made to perform some rational work. This rational work is known as problem in the software parlance. Thus it is very important that we understand how to solve the problem. Under top-down analysis, the problem is broken down into small pieces where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.

Modular Programming - While programming, the code is broken down into smaller group of instructions. These groups are known as modules, subprograms or subroutines. Modular programming based on the understanding of top-down analysis. It discourages jumps using 'goto' statements in the program, which often makes the program flow non-traceable. Jumps are prohibited and modular format is encouraged in structured programming.

Structured Coding - In reference with top-down analysis, structured coding sub-divides the modules into further smaller units of code in the order of their execution. Structured programming uses control structure, which controls the flow of the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

Functional Programming

Functional programming is style of programming language, which uses the concepts of mathematical functions. A function in mathematics should always produce the same result on receiving the same argument. In procedural languages, the flow of the program runs through procedures, i.e. the control of program is transferred to the called procedure. While control flow is transferring from one procedure to another, the program changes its state.

In procedural programming, it is possible for a procedure to produce different results when it is called with the same argument, as the program itself can be in different state while calling it. This is a property as well as a drawback of procedural programming, in which the sequence or timing of the procedure execution becomes important.

Functional programming provides means of computation as mathematical functions, which produces results irrespective of program state. This makes it possible to predict the behavior of the program.

Functional programming uses the following concepts:

First class and High-order functions - These functions have capability to accept another function as argument or they return other functions as results.

Pure functions - These functions do not include destructive updates, that is, they do not affect any I/O or memory and if they are not in use, they can easily be removed without hampering the rest of the program.

Recursion - Recursion is a programming technique where a function calls itself and repeats the program code in it unless some pre-defined condition matches. Recursion is the way of creating loops in functional programming.

Strict evaluation - It is a method of evaluating the expression passed to a function as an argument. Functional programming has two types of evaluation methods, strict (eager) or non-strict (lazy). Strict evaluation always evaluates the expression before invoking the function. Non-strict evaluation does not evaluate the expression unless it is needed.

λ-calculus - Most functional programming languages use λ-calculus as their type systems. λ-expressions are executed by evaluating them as they occur.

Common Lisp, Scala, Haskell, Erlang and F# are some examples of functional programming languages.

Coding style

Programming style is set of coding rules followed by all the programmers to write the code. When multiple programmers work on the same software project, they frequently need to work with the program code written by some other developer. This becomes tedious or at times impossible, if all developers do not follow some standard programming style to code the program.

An appropriate programming style includes using function and variable names relevant to the intended task, using well-placed indentation, commenting code for the convenience of reader and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and error solving easier. Also, proper coding style helps ease the documentation and updation.

Standards and guidelines

Coding Guidelines

Practice of coding style varies with organizations, operating systems and language of coding itself.

The following coding elements may be defined under coding guidelines of an organization:

Naming conventions - This section defines how to name functions, variables, constants and global variables.

Indenting - This is the space left at the beginning of line, usually 2-8 whitespace or single tab.

Whitespace - It is generally omitted at the end of line.

Operators - Defines the rules of writing mathematical, assignment and logical operators. For example, assignment operator '=' should have space before and after it, as in "x = 2".

Control Structures - The rules of writing if-then-else, case-switch, while-until and for control flow statements solely and in nested fashion.

Line length and wrapping - Defines how many characters should be there in one line, mostly a line is 80 characters long. Wrapping defines how a line should be wrapped, if is too long.

Functions - This defines how functions should be declared and invoked, with and without parameters.

Variables - This mentions how variables of different data types are declared and defined.

Comments - This is one of the important coding components, as the comments included in the code describe what the code actually does and all other associated descriptions. This section also helps creating help documentations for other developers.

Documentation guidelines

Software documentation is an important part of software process. A well written document provides a great tool and means of information repository necessary to know about software process. Software documentation also provides information about how to use the product.

A well-maintained documentation should involve the following documents:

Requirement documentation - This documentation works as key tool for software designer, developer and the test team to carry out their respective tasks. This document contains all the functional, non-functional and behavioral description of the intended software.

Source of this document can be previously stored data about the software, already running software at the client's end, client's interview, questionnaires and research. Generally it is stored in the form of spreadsheet or word processing document with the high-end software management team.

This documentation works as foundation for the software to be developed and is majorly used in verification and validation phases. Most test-cases are built directly from requirement documentation.

Software Design documentation - These documentations contain all the necessary information, which are needed to build the software. It contains: **(a)** High-level software architecture, **(b)** Software design details, **(c)** Data flow diagrams, **(d)** Database design

These documents work as repository for developers to implement the software. Though these documents do not give any details on how to code the program, they give all necessary information that is required for coding and implementation.

Technical documentation - These documentations are maintained by the developers and actual coders. These documents, as a whole, represent information about the code. While writing the code, the programmers also mention objective of the code, who wrote it, where will it be required, what it does and how it does, what other resources the code uses, etc.

The technical documentation increases the understanding between various programmers working on the same code. It enhances re-use capability of the code. It makes debugging easy and traceable.

There are various automated tools available and some comes with the programming language itself. For example java comes JavaDoc tool to generate technical documentation of code.

User documentation - This documentation is different from all the above explained. All previous documentations are maintained to provide information about the software and its development process. But user documentation explains how the software product should work and how it should be used to get the desired results.

These documentations may include, software installation procedures, how-to guides, user-guides, uninstallation method and special references to get more information like license updation etc.

Type checking

Type checking • Static type checking means that the correctness of using types is performed at compile-time • Dynamic type checking means that the correctness of using types is performed at run-time

If type checking is performed at compile-time, one may speak about static type checking; otherwise, that is, type checking is performed at run-time, a term dynamic type checking is applied. In principle, type checking can always be performed at run-time if the information about types of values in program is accessible in executing code. Obviously, dynamic type checking, compared to static one, leads to the increase of size and execution time of the object program and lowers the reliability of the compiled code. A programming language is called a language with static type checking or strongly typed language, if the type of each expression can be determined at compile-time, thereby guaranteeing that the type-related errors cannot occur in object program. Pascal is an example of a strongly typed language. However, even for Pascal some checks can be performed only dynamically. For instance, consider the following definitions: table: array [0..255] of char; i: integer; A compiler cannot guarantee that the argument i in the expression table[i] is actually in the specified boundaries, that is, not less than zero and not greater than 255. In some cases, such checking can be performed by techniques like data flow analysis, but it is far from being always possible. It is clear that this sample demonstrates a common situation, namely, the control of slice arguments, occurring in the greater part of programming languages. Of course, this checking is usually performed dynamically.

Scoping rules

The *scope* of a name (variable names, data structure names, procedure names) is the part of the program within which the name can be used. Darwin employs*dynamic scoping*, that is, Darwin does not require that you declare variables before you use them.[3.1] Thus,

```
>x := 5:
>x := x * 99:
>y := z:
```

are perfectly digestible by the Darwin system. This section explores the scoping rules employed by Darwin for variables, structures and routines. Instead of a formal description of the scoping rules Darwin uses, we use a series of examples which covers all cases one would normally encounter when writing programs.
So far all of the names we have defined have been *global* meaning their scope is the entire Darwin environment. There are situations when we do not necessarily want some variables to be *visible* to the entire world. The temporary variables created by procedures are a common example of this; the variables have no meaning outside of the scope of the procedure and should therefore only be accessible from inside that routine. We term such variables *local* variables. Observe the role oftotal in the example below.

```
> new_SetAverage := proc( s : set(real) )
>local total;
>description'This procedure takes a set of real values as an
>argument and calculates the average value in the set';
>total := 0:
>for i from 1 to length(s) do
>total := total + s[i];
>od;
>print('The average is ', total/length(s));
>end:
```

Since variable total is used as a ``scratch'' variable during the computation of the average, no other procedure would require use of it. Therefore, it is advisable to make it local to the routine. To do this, we declare total to be of type local and now it is visible only within the body of procedure NewSetAverage. This explicit declaration is optional in Darwin. If you compare procedure NewSetAverage with the previous versions of SetAverage, you will see that we dynamically definedtotal. What does Darwin do when it encounters a name that has not yet been declared? This is an easy question with a complicated answer. The dogma we suggest you adhere to goes something like as follows:
the best way to learn the Darwin scoping rules is through experimentation; the best way to avoid learning the Darwin scooping rules is through always declaring your local variables.

When you begin to write large programs, subtle scoping problems may arise. These can be particularly difficult to pinpoint and remove. The explicit declaration of a local variable helps to document your code and provides some help for Darwin in determining the intent of your program.

We can use the built-in assigned function to test the scope of a variable. The function
assigned(*t* : name)
takes a parameter t of type name and returns true if t is currently being used as a name in this scope. Otherwise, it returns false.

```
>assigned(total);          # check to see if total is a variable
>                        #   from outside of the procedure
>SetAverage({5, 10, 15});    # call the procedure
>assigned(total);          # check to see if total is a
>                        #   variable now from outside the
>                        #   proc
```

Any local declarations must appear directly after the procedure declaration. They may follow or precede global declarations (see § below) but must precede any option commands (§) or the description command (§). If there is more than one local variable, they must be separated by commas.

```
>example := proc( )
>local x, y, z;
>global a, b, c;
>option polymorphic;
>description'A description of the procedure';
>end:
```

Concurrency mechanisms.

Concurrency in software engineering means the collection of techniques and mechanisms that enable a computer program to perform several different tasks simultaneously, or apparently simultaneously. The need for concurrency in software first arose in the very early days of computing. Although early computers were very much slower than modern machines, their peripheral devices (such as paper tape or punched-card readers for input, and teletypewriters for output) were much slower still, and it was soon recognized that many programs could be made to run very much faster if they did not spend so much time waiting for input/output (I/O). The solution is easy in principle—simply allow I/O transfers to be performed concurrently with each other and with normal computation (i.e., computation involving only the central processor). To achieve this, new features needed to be added to both hardware and software.

Today, such features—concurrency mechanisms, the most important of which are discussed in this article—are commonplace and form a vital part of all modern computer systems. The operating system of any computer (e.g., Windows, UNIX, Linux) makes use of concurrency to enable the user to do several different things simultaneously or to allow many different users to access the computer at the same time. Many operating systems allow a large number of tasks to be run concurrently and the simultaneous use of peripheral devices. The earliest applications of concurrency were in real-time systems such as operating systems, in which the software must perform actions at times determined by external events.

Unit-5 :

Quality assurance

Quality assurance is "a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements" (IEE83). The purpose of a software quality assurance group is to provide assurance that the procedures, tools, and techniques used during product development and modification are adequate to provide the desired level of confidence in the work products.

Often, software quality assurance personnel are organizationally distinct from the software development group. This adds a degree of impartiality to quality assurance activities, and allows quality assurance personnel to become specialists in their discipline. In some organizations, quality assurance personnel function in an advisory capacity, while in others the quality assurance group actively develops standards, tools, and techniques, and examines all work products for conformance to specifications.

Preparation of a Software Quality Assurance Plan for each software project is a primary responsibility of the software quality assurance group. Topics in a Software Quality Assurance Plan include (BUC79):

1. Purpose and scope of the plan
2. Documents referenced in the plan
3. Organizational structure, tasks to be performed, and specific responsibilities as they relate to product quality.
4. Documents to be prepared and checks to be made for adequacy of the documentation
5. Standards, practices, and conventions to be used
6. Reviews and audits to be conducted
7. A configuration management plan that identifies software product items, controls and implements changes, and records and reports changed status
8. Practices and procedures to be followed in, reporting, tracking, and resolving software problems .
9. Specific tools and techniques to be used to support quality assurance activities
10. Methods and facilities to be used to maintain and store controlled versions of identified software
11. Methods and facilities to be used to protect computer program physical media
12. Provisions for ensuring the quality of vendor-provided and subcontractor-developed software
13. Methods and facilities to be used in collecting, maintaining, and retaining quality assurance records

Other duties performed by quality assurance personnel include:

1. Development of standard policies, practices, and procedures
2. Development of testing tools and other quality assurance aids
3. Performance of the quality assurance functions described in the Software Quality Assurance Plan for each project
4. Performance and documentation of final product acceptance tests for each software product.

More specifically, a software quality assurance group may perform the following functions: *(*

1. During analysis and design, a *Software Verification Plan* and an *Acceptance Test Plan* are prepared. The verification plan describes the methods to be used in verifying that the requirements are satisfied by the design documents and that the source code is consistent with the requirements specifications and design documentation. The *Source Code Test Plan* (discussed below) is an important component of the *Software Verification Plan.* The *Acceptance Test Plan* includes test cases, expected outcomes, and capabilities demonstrated by each test case. Often, quality assurance personnel will work with the customer to develop a single *Acceptance Test Plan. In* other cases, the customer will develop an *Acceptance Test Plan* independent of the *Quality Assurance Plan. In either* case, an in-house *Acceptance Test Plan* should be developed by quality assurance personnel. Following completion of the verification plan and the acceptance plan, a Software Verification Review is held to evaluate the adequacy of the plans.

2. During product evolution, In-Process Audits are conducted to verify consistency and completeness of the work products. Items to be audited for consistency include interface specifications for hardware, software, and people; internal design versus functional specifications, source code versus design documentation; and functional requirements versus test descriptions. In practice, only certain critical portions of the system may be subjected to intensive audits.
3. Prior to product delivery, a Functional Audit and a Physical Audit are performed. The Functional Audit reconfirms that all requirements have been met. The Physical Audit verifies that the source code and all associated documents are complete, internally consistent, consistent with one another, and ready for delivery. A *Software Verification Summary* is prepared to describe the results of all reviews, audits, and tests conducted by quality assurance personnel throughout the development cycle.

5.1.3. Types of Test

There are four types of tests that the source code must satisfy: function tests, performance tests, stress tests, and structure tests. Function tests and performance tests are based on the requirements specifications; they are designed to demonstrate that the system satisfies its requirements. Therefore, the test plan can be only as good as the requirements, which in turn must be phrased in quantified, testable terms.

Functional test cases specify typical operating conditions, typical input values, and typical expected results. Function tests also test behavior just inside, on, and just beyond the functional boundaries. Examples of functional boundary tests include testing a real-valued square root routine with small positive numbers, zero, and negative numbers; or testing a matrix inversion routine on a one-by-one matrix and a singular matrix.

Performance tests are designed to verify response time under varying loads, percent of execution time spent in various segments of the program, throughput, primary and secondary memory utilization, and traffic rates on data channels and communication links. ,

Stress tests are designed to overload a system in various ways. Examples, of stress tests include attempting to sign on more than the maximum number of allowed terminals, processing more than the allowed number of identifiers or static levels, or disconnecting a communication link.

Structure tests are concerned with examining the internal processing logic of a software system. The particular routines called and the logical paths traversed through the routines are the objects of interest. The goal of structure testing is tc traverse a specified number of paths through each routine in the system tc establish thoroughness of testing.

Each test case in the *Source Code Test Plan* should provide the following information: Type of test (function, performance, stress, structure) \
Machine configuration
Test assumptions
Requirements being tested
Exact test stimuli
Expected outcome

Walk through and Inspection

The two primary human testing methods are code inspections and walkthroughs. Since the two methods have a lot in common, we will discuss their similarities together here. Their differences are discussed in subsequent sections.
Inspections and walkthroughs involve a team of people reading or visually inspecting a program. With either method, participants must conduct some preparatory work. The climax is a "meeting of the minds," at a participant

conference. The objective of the meeting is to find errors but not to find solutions to the errors. That is, to test, not debug.

Code inspections and walkthroughs have been widely used for some time. In our opinion, the reason for their success is related to some of the principles. In a walkthrough, a group of developers—with three or four being an optimal number—performs the review. Only one of the participants is the author of the program. Therefore, the majority of program testing is conducted by people other than the author, which follows the testing principle stating that an individual is usually ineffective in testing his or her own program.

An inspection or walkthrough is an improvement over the older desk-checking process (the process of a programmer reading his or her own program before testing it). Inspections and walkthroughs are more effective, again because people other than the program's author are involved in the process.

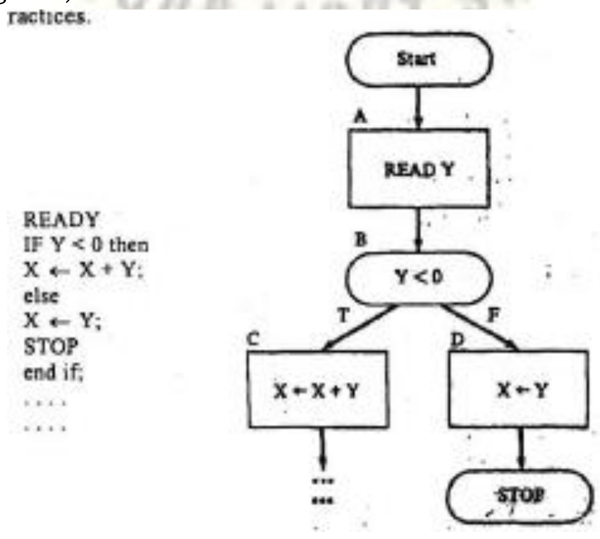
Static analysis

Static analysis is a technique for assessing the structural characteristics of source code, design specifications, or any notatipnal representation that conforms to well-defined syntactic rules. The present discussion is restricted to static analysis of source code. In static analysis, the structure of the code is examined, but the code is not executed. Because static analysis is concerned with program structure, it is particularly useful for discovering questionable coding practices and departures from coding standards, in addition to detecting structural errors such as uninitialized variables and mismatches between actual and formal parameters.

Static analysis can be performed manually using walkthrough or inspection techniques; however, the term "static analysis" is most often used to denote examination of program structure by an automated tool (RAM75, MIL75,, FOS76, SOF80, GRC83). A static analyzer will typically construct a symbol table and a graph of control flow for each subprogram, as well as a call graph for the entire program. The symbol table contains information about each variable: its type attributes, the statement where declared, statements where set to a new value, and statements where used to provide values.

Figure 5.1 illustrates a code segment and the corresponding control flow graph. The nodes in a control-flow graph correspond to basic blocks of source code, and the arcs represent possible transfers of control between blocks. A basic block of source code has the property that if the first statement in the block is executed, every statement in the block will be executed. In a call graph, the nodes represent program units and the arcs represent potential invocation of one program unit by another.

Given a control-flow graph and a symbol table that contains, for each variable in a subprogram, the statement numbers where the variables are declared, set, and used, a static analyzer can determine data flow information such as uninitialized variables (on some control paths or on all paths), variables that are declared but never used, and variables that are set but not subsequently used



```
READY
IF Y < 0 then
X ← X + Y;
else
X ← Y;
STOP
end if;
....
....
```

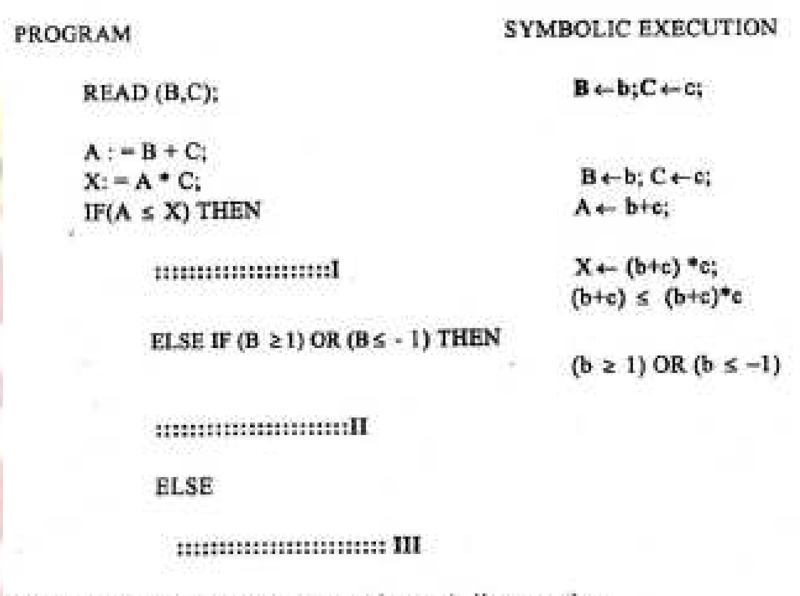
Figure 5.1 A code segment and control flow graph

(on some paths on all paths). For example, in Figure 5.1 - variable X is uninitialized on path ABC and is set but not used on path ABD.

Static analyzers typically produce lists of errors, questionable coding practices.(anomalies), and departures from coding standards. For instance, a variable that is uninitialized on all paths is a structural error. A variable that is set but not used on any subsequent control path, or a variable that is declared but never used, is not an error, but it is an anomaly that may be symptomatic of an error. Departures from coding standards, such as using non-ANSI FORTRAN constructs, backward GOTO transfers of control, or jumps into loop bodies, can also be detected by static analysis.

Symbolic Exception

Symbolic execution is a validation technique in which the input variables of a program unit are assigned symbolic values rather than literal values. A program is analyzed by propagating the symbolic values of the inputs into the operands in expressions. The resulting symbolic expressions are simplified at each step in the computation so that all intermediate computations and decisions are always expressed in terms of the symbolic inputs. For instance, evaluation- of an. assignment statement results in association of a symbolic expression with the left-hand variable. When that variable is used in subsequent expressions, the current symbolic value is used. In this manner, all computations and decisions arc expressed as symbolic values of the inputs.

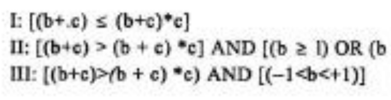


```
PROGRAM                                      SYMBOLIC EXECUTION

    READ (B,C);                                  B ← b;C ← c;

    A : = B + C;                                  B ← b; C ← c;
    X: = A * C;                                   A ← b+c;
    IF(A ≤ X) THEN

         :::::::::::::::::::::I                    X ← (b+c) *c;
                                                  (b+c) ≤ (b+c)*c
         ELSE IF (B ≥ 1) OR (B ≤ - 1) THEN
                                                  (b ≥ 1) OR (b ≤ −1)

         :::::::::::::::::::::::II

         ELSE

          ::::::::::::::::::::::::: III
```

Figure 5.2 A Program segment and its symbolic execution.

Figure 5.2 illustrates a program segment and its symbolic execution.

predicates from the IF statements can be conjoined to form path conditions that describe the constraints under which various segments of code will be executed. Figure 5.3 illustrates path conditions for program segments I, II, and III in Figure, 5.2 Figure 5.4 is a plot of the boundaries between regions I, II, and III as functions of symbolic input values b and c. Any literal values of b and c chosen from region I, II, or III will result in execution of code segment I, II, or III.



I: [(b+.c) ≤ (b+c)*c]
II: [(b+c) > (b + c) *c] AND [(b ≥ l) OR (b ≤−1)l
III: [(b+c)>(b + c) *c) AND [(−1<b<+1)]

Figure 5.3 Symbolic path expressions

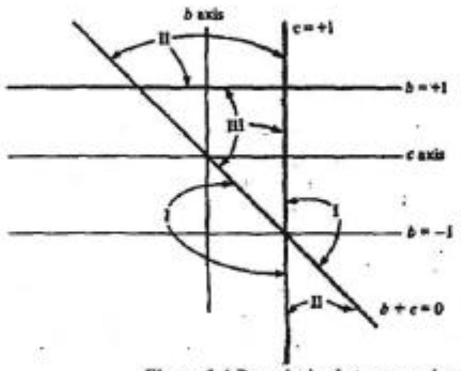


Figure 5.4 Boundaries between regions I, II and III.

Symbolic execution can thus be used to derive path conditions that can be solved to find input data values that will drive a program along a particular execution path, provided all predicates in the corresponding path condition are linear and functions of the symbolic input values. When the predicates are nonlinear in the input values, the path condition may or may not be solvable because systems of nonlinear inequalities are in general unsolvable.

Unit testing and Debugging

Static analysis is used to investigate the structural properties of source code. Dynamic test cases are used to investigate the behavior of source code by executing the program on the test data. As before, we use the term "program unit" to denote a routine or a collection of routines implemented by an individual programmer. In a well-designed system, a program unit is a stand-alone program or a functional unit of a larger system.

5.1.6.1 Unit Testing

Unit testing comprises the set of tests performed by an individual programmer prior to integration of the unit into a larger system. The situation is illustrated as follows:

Coding & debugging >unit testing > integration testing

A program unit is usually small enough that the programmer who developed it can test it in great detail, and certainly in greater detail than will be possible when the unit is integrated into an evolving software product.

There are four categories of tests that a programmer will typically perform on a program unit:

Functional tests
Performance tests
Stress tests
Structure tests

Functional test cases involve exercising the code with nominal input values for which the expected results are known, as well as boundary values (minimum values, maximum values, and values on and just outside the functional boundaries) and special values, such as logically related inputs, lxl matrices, the identity matrix, files of identical elements, and empty files.

Performance testing determines the amount of execution time spent in various parts of the unit, program throughput, response time, and device utilization by the program unit. A certain amount of performance tuning may be done during unit testing; however, caution must be exercised to avoid expending too much effort on fine tuning of a program unit that contributes little to the overall performance of the entire system. Performance testing is most productive at the subsystem and system levels.

Stress tests are those tests designed to intentionally break the unit. A great deal can be learned about the strengths and limitations of a program by examining the manner in which a program unit breaks.

Structure tests are concerned with exercising the internal logic of a program and traversing particular execution paths. Some authors refer collectively to functional, performance, and stress testing as "black box" testing, while structure testing is referred to as "white box" or "glass box" testing. The major activities in structural testing are deciding which paths to exercise, deriving test data to exercise those paths, determining the test coverage criterion to be used, executing the test cases, and measuring the test coverage achieved when the test cases are exercised.

A test coverage (or test completion) criterion must be established for unit testing, because program units usually contain too many paths to permit exhaustive testing. This can be seen by examining the program segment in Figure 5.4 .a. As

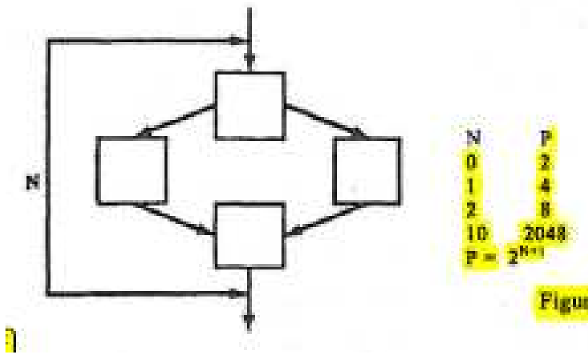


Figure 5.4.a Loop paths

Debugging

Debugging is the process of isolating and correcting the causes of known errors. Success at debugging requires highly developed problem-solving skills; Commonly used debugging methods include induction, deduction, and backtracking (BR073, MYE79). Debugging by induction involves the following steps:

1. Collect the available information. Enumerate known facts about the observed failure and known facts concerning successful test cases. What are the observed symptoms? When did the error occur? Under what conditions did it occur? How does the failure case differ from successful cases?
2. Look for patterns. Examine the collected information for conditions that differentiate the failure case from the successful cases.
3. Form one or more hypotheses. Derive one or more hypotheses from the observed relationships. If no hypotheses are apparent, re-examine the available information and collect additional information, perhaps by running more test cases. If several hypotheses emerge, rank them in the order of most likely to least likely.
4. Prove or disprove each hypothesis. Re-examine the available information to determine whether the hypothesis explains all aspects of the observed problem. Do not overlook the possibility that there may be multiple errors. Do not proceed to step 5 until step 4 is completed.

5. Implement the appropriate corrections. Make the correction (s) indicated by your evaluation of the various hypotheses. Make the corrections to a backup copy of the code, in case the modifications are not correct.
6. Verify the correction. Rerun the failure case to be sure that the fix corrects the observed symptom. Run additional test cases to increase your confidence in the fix. *Rerun* the previously successful test cases to be sure that your fix has not created new problems. If the fix is successful, make the fix-up copy the primary version of the code and delete the old copy. If the fix is not successful, go to step 1.

Debugging by deduction proceeds as follows:

1. List possible causes for the observed failure.
2. Use the available information to eliminate various hypotheses.
3. Elaborate the remaining hypotheses.
4. Prove or disprove each hypothesis.
5. Determine the appropriate corrections.
6. Verify the corrections.

Debugging by backtracking involves working backward in the source code from the point where the error was observed in an attempt to identify the exact point where the error occurred. It may be necessary to run additional test cases in order to collect more information. Techniques for collecting the necessary information are described in the following paragraphs.

Traditional debugging techniques utilize diagnostic output statements, snap-shot dumps, selective traces on data values and control flow, and instruction-dependent breakpoints. Modern debugging tools utilize assertion-controlled breakpoints and execution histories.

Diagnostic output statements can be embedded in the source code as specially formatted comment statements that are activated using a special translator option. Diagnostic output from these statements provides snapshots of selected components of the program state, from which the programmer attempts to .infer program behavior. The program state includes the values associated with all currently accessible symbols and any additional information, such as the program stack and program counter, needed to continue execution from a particular point in the execution sequence of the program.

A snapshot dump is a machine-level representation of the partial or total program state at a particular point in the execution sequence. A structured snapshot dump is a source-level representation of the partial or total state of a program. The names and current values of symbols accessible at the time of the snapshot are provided.

A trace facility lists changes in selected state components. In its simplest form, a trace will print all changes in data values for all variables and all changes in control flow. A selective trace will trace specific variables and control flow in specific regions of the source text.

A traditional breakpoint facility interrupts program execution and transfers control to the programmer's terminal when execution reaches a specified "break" instruction in the source code. The programmer can typically examine the program state, change values of state components, set new breakpoints, and resume execution in either normal mode or single step mode, perhaps starting at another x instruction location.

System testing

System testing involves two kinds of activities: integration testing and acceptance testing. Strategies for integrating software components into a functioning product include the bottom-up strategy, the top-down strategy, and the sandwich strategy. Careful planning and scheduling are required to ensure that modules will be available for integration into the evolving software product when needed. The integration strategy dictates the order in which modules must be available, and thus exerts a strong influence on the order in which modules are written, debugged, and unit tested.

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests to verify that the implemented system satisfies its requirements. Acceptance tests are typically performed by the quality assurance and /or customer organizations. Depending on local circumstances, the development.

5.1.7.1 Integration Testing

Bottom-up integration is the traditional strategy used to integrate the components of a software system into a functioning whole. Bottom-up integration consists of unit testing, followed by subsystem testing, followed by testing of the" entire system. Unit testing has the goal of discovering errors in the individual modules of the system. Modules are tested in isolation from one another in an artificial environment known as a "test harness," which consists of the driver programs and data necessary to exercise "the modules. Unit testing should be as exhaustive as possible to ensure that each representative case handled by each module has been tested. Unit testing is eased by a system structure that is composed of small, loosely coupled modules.

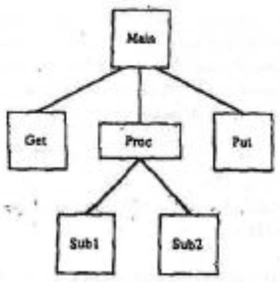
A subsystem consists of several modules that communicate with each other through well-defined interfaces. Normally, a subsystem implements a major segment of the total system. The primary purpose of subsystem testing is to verify operation of the interfaces between modules in the subsystem. Both control and data interfaces must be tested. Large software systems may require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. In most software systems, exhaustive testing of subsystem capabilities is not feasible due to the combinational complexity of the module interfaces; therefore, test cases must be carefully chosen to exercise the interfaces in the desired manner.

System testing is concerned with subtleties in the interfaces, decision logic, % control flow, recovery procedures, throughput, capacity, and timing characteristics of the entire system. Careful test planning is required to determine the extent and nature of system testing to be performed and to establish criteria by which the results will be evaluated.

Disadvantages of bottom-up testing include the necessity to write and debug test harnesses for the modules and subsystems, and the level of c mplexity that results from combining modules and subsystems into larger and larger units. The extreme case of complexity results when each module is unit tested in isolation and all modules are then linked and executed in one single integration run. This is the "big bang" approach to integration testing. The main problem with big-bang integration is the difficulty of isolating the sources of errors.

Test harnesses provide data environments and calling sequences for the routines and subsystems that are being tested in isolation. Test harness preparation can amount to 50 percent or more of the coding arid debugging effort for a software product.

Top-down integration starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level "skeleton" has been thoroughly tested, it becomes the test harness for its immediately subordinate routines. Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.



1. TEST MAIN; STUBS FOR GET, PROC, PUT
2. ADD GET; TEST MAIN, GET
3. ADD PROC; STUBS FOR SUB 1 .SUB 2
4.ADD PUT; TEST MAIN, GET, PROC, PUT
5. ADD SUB1
   TEST MAIN, GET, PROC, PUT, SUB1
6. ADDSUB2
   TEST MAIN, GET, PROC, PUT, SUB1, SUB2

Figure 5.5 Top-down integration testing strategy.

Figure 5.5 illustrates integrated top-down integration testing. Top-down integration offers several advantages:

1. System integration is distributed throughout the implementation phase. Modules are integrated as they are developed.
2. Top-level interfaces are tested first and most often.
3. The top-level routines provide a natural test harness. for lower-level routines.
4: Errors are localized to the new modules and interfaces that are being added.

Formal verification:

Formal verification involves the use of rigorous, mathematical techniques to demonstrate that computer programs have certain desired properties. The methods of input-output assertions, weakest preconditions, and structural induction are three commonly used techniques. Each is discussed to turn.

5.1.8.1 Input-Output Assertions

The method of input-output assertions was introduced by Floyd (FL067) and refined by Hoare (HOA73) and Dijkstra (DU76). Floyd's work was seminal to the entire field of formal verification.

Using input-output assertions, predicates (assertions) are associated with the entry point, the exit point, and various intermediate points in the source code. The predicates, or verification conditions, must be true whenever the associated code is executed. The notation (P) S (R) is used to mean that if predicate P is true prior to executing code segment S, predicate R will be true following execution of S. For example,

(1< i < N) i : = i + 1 (2 < i < N+1) The composition rule of logic permits conjunctions of predicates to be formed along particular execution paths:
(P) S 1, (Q) and (Q), S2 (R) implies (P) S 1; S2 (R)

The composition rule permits the following statement: If all the intermediate predicates are true along a particular execution path, the truth of the input assertion (input predicate) will imply the truth of the output assertion (output predicate) for that execution path.

The method of input-output assertions states that if the conjunction of predicates from the input assertion to an output assertion is true, and if the input assertion is satisfied by the input conditions, and if the program terminates after following the execution path of interest, then the output assertion will be true on termination of the program.

Termination is proved by showing that the execution sequence for each loop monotonically decreases (increases) some nonnegative (negative) property on each pass through the loop. Due to the characteristics of the loop, this property must eventually reach a lower (upper) bound, and execution of the loop will terminate. For example, an integer loop counter starts at N > 0 and is decremented by 1 until it reaches 0, or it starts at 0 and is incremented by 1 until it reaches N>0.

An example from Floyd's original paper, rewritten in Ada, is illustrated in Figure 5.6 (which is similar to Figure 5.4)

```
Function SUM (A: INT ARRAY; N: NATURAL) return INTEGER is - N e
J + (J + is the set of positive integers)
I. NATURAL: = 1;
- (N G J +) and (I = 1)
S: INTEGER: =0;
- (N e J +) and (I = 1) and (S = 0)
while I < = N loop
- (N e J +) and (I e J +) and (K = N + 1) and S = Z Aj S: = S + A (I):
- (N e J +) and (I e J +) and (K = N + 1) and S = Z Aj
I: = I+ 1;
- (N e J +) and (I G J +) and (2 < = K = N + 1) and S = Z Aj
J-I
end loop;
1-1 (N+11-1
- (N e J +) and (I = N + 1) and S = Z Aj = Z Aj
J=I ]=i
N
i.e.; S = £ Aj
return (S);
```

Figure 5.6 An Ada function with intermediate assertions.

In practice, it is not necessary to list all the intermediate predicates illustrated in Figure 8.11. The minimal requirement is that a predicate be associated with each innermost nested loop. In Figure 8.11, the loop predicate is

1-1 S=ZAj

Loop predicates must be shown to be invariant relations; i.e., a loop invariant must be true independent of the number of times the loop is traversed. In particular, a loop invariant must satisfy the following conditions:

It must be true on loop entry.
It must be true independent of the number of loop traversals.
It must imply the desired condition on loop exit.
. The loop invariant in Figure 8.11 satisfies these conditions. It is true on loop entry, when S = 0 and 1=1:
S = Z Aj = 0
j-i
and it implies the desired output predicate, when I = N + 1:
->'" (N+9-1 N
S = Z Aj = Z Aj

Enhancing maintainability during development

Many activities performed during s/w development enhance the maintainability of a s/w product. Some of these activities are:
5.2.1.1 Analysis activities

The analysis phase s/w development is concerned with determining customer requirements and constraints and establishing feasibity of the product. From the maintanance viewpoint, the most important activities that occur during analysis are establishing standards and guidelines for the projects and the work products to ensure uniformly of the products. Setting of milestones to ensure that the work products are produced on schedule specifying quality assurance procedures, luenurying produci enhancements that will most likely occur following initial delevery of the item and estimating the resources (personnel, equipment, floor space) required to perform maintenance activities.

5.2.1.2 Design activities

The most most important activity for ecnhansing maintainability- during architectural design is to emphasize clarity, modularity and ease of modification as the primary design criteria. From various ways of structuring system, the designers will choose a particular structure on the basis of certain design criteria sucn as coupling and cohesion of modules, efficiency consideration etc. The forms of design documentation aid the s/w main^ainer who must understand the s/w product well enough to modify it and revalidate it.

Detailed design in concerned with specifying algorithmic details, concrete data representations and details of the interfaces among routines and data structures. The activities for enhancing maintainability during this phase are: Standardize notations should be used to describe algorithms, data structures and procedure interface specifications. Side effects and exception handling for each routine should be specified, cross-reference directories should be provided for easy modification.

5.2.1.3 Implementation activities

The primary goal of implementation is to produce the s/w that is easy to understand and easy to modify. The activities for enhancing maintainability during this phase are: Single entry, single exit coding constructs should be used standard identation of constructs should be observed, a straight forward coding style must be adapted ease of-maintanance is enhanced by use of symbolic constraints to parameterize the s/w; standard documentation prologues for each routine should be provided, !

5.2.1.4 Other activities

There are two particularly important supporting documents that should be prepared during s/w development life cycle in order to ease maintanance activities. The documents are:

(i) Maintenance guide-provides a technical description of the operational capabilities of the entire-system and hierarchy diagrams, call graphs and cross reference directories for the system.
(ii) A Test-suit - This is a file of test cases
This should contain a set of test data and actual results from those tests. When s/w is modified test cases are added to the test suite to validate the modifications.
Documentation for the test suite should specify the system configuration assumpt-- and conditions for each test case, the actual input data for each testand a description of expected results for each test. This test suite documentation should be provided.

Managerial aspects of software maintenance

Successful s/w product requires a combination of manegerial skills and technical expertise. In this section we are going to discuss some managerial concerns of s/w maintanance.

One of the most important aspects of s/w maintanance involves tracking and control of maintanance activities. Maintanance activity for a s/w product usually occurs in responce to change request filed by the user of the product. Change request processmg; can be described by the following algorithm.

```
-software change request initiated.
-request anlaysed
if (request not valid) then
        -request closed
else
-request & recommdendations submitted to change control board.
if (change control board concurs) then
        -modifications performed with priority and constraints established by change control board.
        -regression tests performed -changes submitted to change control boards.
if (changewntrol board approves) then
        -master tape updated -external documentation updated
        -update distributed as directed by change control board
else
        -control board objections satisfied and changes resubmitted to control board
else
        -request closed.
```

Change requests are usually initiated by users. A change request may entail enhancement, adaptation or error correction. A change request is first reviewed by an analyst. In some cases, the request may report a user problem that is not caused by the s/w being maintained. In this case, the analyst notifies the user and with the concurrence of the user, closes the request. Otherwise the analyst submits to the control board the change request, the proposed fix, and an estimate of the resourses required to satisfy the request.

Change Control board

This reviews and approves change requests. Approved changes are forwarded to the maintanance programmers for action. The s/w is modified, revalidated and submitted to the change control board for approval.

If the control board approves, the master taps and external documents are updated and the modified s/w is distributed to user sites. If the control board does not approve, the control board objections are addressed by the analysts and programmer for review it.

Change request Summaries

The status of change requests and s/w maintenance activities should be summarized on a weekly or monthly basis. The summary should report emergency problems. In addition, a maintenance trends summary should be included in each change request summary, a trends summary is a graph showing the number of new requests and the total number of open requests as a function of time.

Quality assurance activities

The primary function of a quality assurance group during s/w maintenance is to ensure the s/w quality does not degrade as a result of maintenance activities. In particular, the quality assurance group should conduct audits and spot checks to determine that external
documents are properly updated to reflect modifications.

Organizing maintenance programmers

Software maintenance can be performed by the development team or by members of a seperate organization. The advantages of development team are: Members of the development team will be intimately familiar with the product. They will take great care to design and implement the system to enhance maintainability. The disadvantages are: Maintenance acitivities may divert the developers from their new project. Maintenance by the developers make the maintenance activity vulnerable to personnel turnover.
Maintenance by a seperate group forces more attention to standards and high quality documentation. The advantae of this is releasing the development team to purpose other activities.

Each approach has its own advantages. A desirable method of organising team for maintenance is to periodically rotate programmers between development and maintenance.

Configuration management

This is concerned with trading and controlling of the work products that constitute a s/ w product. During s/w maintenance a configuration management plan and configuration management tools are required to track and control various work versions of work products that constitute a s/w product. Tracking and controlling multiple versions of a s/w product is a significant issue in s/w maintenance.

Configuration management data base : s/w tools to support configuration management include configuration management data base and vession control library systems.

A configuration management database can provide information concerning product structure current revision number current status and change request history for each product version.
A configuration management database should be able to answer:
How many versions of each product exist?
How do the version differ?
Which versions of which products are distributed to which sites?
What documents are available for each version of each product?
What is revision history of each component of each version of each product?
When will the next revision of a given version of a product willbe available?
What hardware configuration is required to operate a specific version of the product?
Which revisions constitute a specific version of the product?
Which version are affected by a given component revision?
Which version are affected by a specific error report?
Which errors were corrected in a specific error report?
How many errors were reported/fixed in the past month?

What are the causes of reported product failure?
Which components are functionally similar in which version?
Is a given old version still used by some users?
What was the configuration of a given version on a given date?

Version control libraries

A configuration managememsdatabase provides a macro view of a product family, while a version control library controls the various files that constitute the various versions of a s/w product. Version control library system is not a database and does not contain the information required to answer questions such as above. Entities in a version control library may include source code, relocatable object code, job control commands, data files and supporting documents. Each entity in a library must carry an identity stamp that includes a version number, data, time and programmer identity. Operation that can be performed on a library include creation of the library, addition and deletion of components, preparation of back-up copies, editing of files, listing of summary statistics and compilation / assembly of specified version of the system.

One example for version control library is the Revision control system (RCS). RCS treats all files as text. Using RCS different versions of text files can be created, stored and retrieved. Changes to versions are logged and changes can be merged to create versions.

Effective configuration management systems, integrate support tools such as configuration management databases and version control library systems within the managerial frame work of change control.

Source Code metrics

Most of the metrics incorporate easily computed properties of the source code such as the number of operators and operands the complexity of the control flow graph, the number of parameters and global variables in routines and the number of levels and manner of interconnection of the call graph.

An overall measure of software complexity, by whatever technique, must account for factors such as the computing environment, the application area, the particular algorithms implemented, the required levels of reliability and efficiency and the characteristics of product users.

Two source code metrics are discussed here:
(1) Halstead's effort equation and (2) Meccabe's cyclomatic complexity measure.

5.2.4.1 Halstead's effort equation

Halstead developed a number of metrics that are computed from easily obtained properties of the source code. These properties include:
Nl - the total number of operators in a program.
N2 - the total number of operands in the program.
nl - the number of unique operations int he program.
n2 - the number of unique operands in the program.
Halstead defines several quantities from the above. For example, Hal stead's estimator of program length is: N = nl log2 nl + n2 log2 n2. Program volume is defined as: Program effort is defined as: V/IL.
(ie) E= (nl * n2 ((N1+N2) * log2(nl+n2)/(2 * n2)

An interesting application of Halstead's metrics is detecting similarity of programs.

5.2.4.2 Meccabe's cyclomatic metrics

Meccabe has observed that the difficulty of understanding a program is largely determined by the complexity of the control flow graph for that program. The cyclomatic number V of a connected graph G is the number of linearly independent paths in the graph. V(a) is computed as follows:
V(a) = E*n + 2p, where,

E is the number of edges.
n is the number of nodes.
p is the number of connected components.
For example, consider the following graph,
Jr? - ^{S} Y°^{r},! ^{h}l$^{S \text{ graph}}$,' The dashed Hne in the above g^{ra}P^{h} connecting the output node f to input node 'a' is added to produce a connected graph.

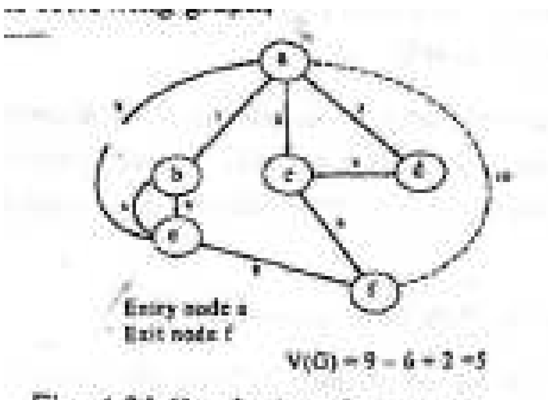


Fig. 4.21 Comlexity of control flow

Meccabe observes that for a structured program with single entry, single exit constructs V equals the number of predicates plus one. Also, provided G is planer V is equal to the number of resions in G.

During the implementation phase, one might use source-code metrics to identify routines that are candidates for further refinement and rewriting. During maintenance, the complexity metrics can be used to track and control the complexity level of modified routines.

Other maintenance tools and techniques.

Automated tools to support software maintenance include technical support tools and managerial tools to support the technical aspects of software maintenance include analysis and design tools, implementation tools and debugging and tesing tools. Automated tools of special importance for software maintenance include text editors, debugging aids, cross-reference generators, linkage editors, comparators, complexity metric calculator, version
control systems and configuration management data bases.

A text editor permits rapid, efficient modification of source programs, test data supporting documents. Test editors can be used to insert and replace segments source code, internal comments, test data ad supporting documents to systematically change all occurances of an identifiers or other textual string, to locate all references to a given identifier or other string of text and to save both old and new versions of a routine, test file of document. A syntax directed text editor can be used to preserve the structure of source code and an intelligent text editor can ensure that all cross references in the supporting documents are correctly updated.

Debugging aids provide traps, dumps, traces, assertion checking and history files to aid in locating the causes of known errors. System level cross-reference generators provide cross-reference listings for procedure calls, statement usage and data references. Cross reference directories provide the calling structure the 'procedure names and statement numbers where formal parameters, local variables and global variables are defined, set and used.

A linkage editor links together object modules of compiled code to produce an executable program. These can be used by a maintenance programmer to configure a system in various ways and to link selectivity recompiled modules into a software system.

A comparator compares tv.o files of information and reports the differences. Comparators can be used during maintenance to compare/two versions of source program, a test suite, a file of test results or two versions of a supporting documents. This allows to pinpoint the differences between versions, to determine whether a modification has achieved a desired result and to determine whether adverse side effects have been introduced by a modification.

Use of complexity metrics requires automated tools to compute the measure of interest. The complexity of source code can be compared before and after a modification is made to determine whether complexity has increased as a result of the modification.

We have already discussed, version control system and configuration management data base and how they are used in software maintenance.
Other aspects of change control that can be supported by automated tools include change request, processing, periodic status reporting change control board recommendations, quality assurance, tracking and updating of historical data.

*****