SOFTWARE ENGINEERING
(SEE6G)

SYLLABUS

Title of the Course/ Paper	**SOFTWARE ENGINEERING**		
Core	**III Year & Sixth Semester**	Credit: 5	
Objective of the course	This course introduces the details about the concepts of life cycle of software		
Course outline	Unit 1: Introduction to Software Engineering Some definition – Some size factors – Quality and productivity factors – Managerial issue. Planning a Software Project: Defining the problem – Developing a solution strategy – planning the development process – planning an organization structure – other planning activities.		
	Unit-2: Software Cost Estimation: Software – Cost factors – Software cost estimation techniques – specification techniques – level estimation – estimating software maintenance costs. The software requirements specification – formal specification techniques - languages and processors for requirements specification.		
	Unit 3: Software Design: Fundamental Design concepts – Modules and modularizing Criteria – Design Notations – Design Techniques – Detailed Design Consideration – Real time and distributed system design – Test plan – Mile stones walk through and inspection.		
	Unit-4: Implementation issues : Structured Coding techniques – coding style – standards and guidelines – documentation guidelines – type checking – scooping rules – concurrency mechanisms.		
	Unit-5 : Quality assurance – walk through and inspection - Static analysis – symbolic exception – Unit testing and Debugging – System testing – Formal verification: Enhancing maintainability during development – Managerial aspects of software maintenance – Configuration management – source code metrics – other maintenance tools and techniques.		

1. Recommended Texts

i. Richard E.Fairly - Software Engineering Concepts - Tata McGraw-Hill book Company.

2. Reference Books

i. R.S.Pressman, 1997, Software Engineering – 1997 - Fourth Ed., McGraw Hill.
ii. Rajib Mall ,2004,Fundamentals of Software Engineering,2nd Edition, PHI.

UNIT I

Definition :Software Engineering

"Software engineering is the technological and managerial discipline concerned with systematic production of software products developed and modified on time and within cost estimate"

The primary goals of software engineering are to improve the quality of software products and to increase the productivity and job satisfaction of the software engineers.

SOME DEFINITIONS

Programmer is the person concerned with the details of implementing, packaging and modifying algorithms and data structures written in particular programming languages. He is also concerned with the issues of analysis, design, verification and testing, documentation and software maintenance, and project management.

The computer software or the software product includes source code and all the associated documents and documentation that constitutes software product.

Documentation explains the characteristics of the documents. Internal documentation explains the characteristics of the code and external documentation explains the characteristics of the documents associated with the code.

Developer is the software engineer who develops the software product.

Customer is the individual or organization that initiates procurement or modification of software product.

Software quality attributes include usefulness, clarity, reliability, efficiency, and cost effectiveness.

Software reliability is the ability of a program to perform a required function under stated conditions for a stated period of time.

Real-time systems are the software products implemented on microprocessors critically constrained in memory space and execution time.

A software product must be cost-effective in development and maintenance.

SOME SIZE FACTORS

1. **Total effort devoted to software**
   - As computing systems become more numerous, the demand for high quality software increases at an increasing rate.
   - Current demand for software technologists exceeds the available supply.
   - Software maintenance is rapidly increasing because with passing time more software accumulates.

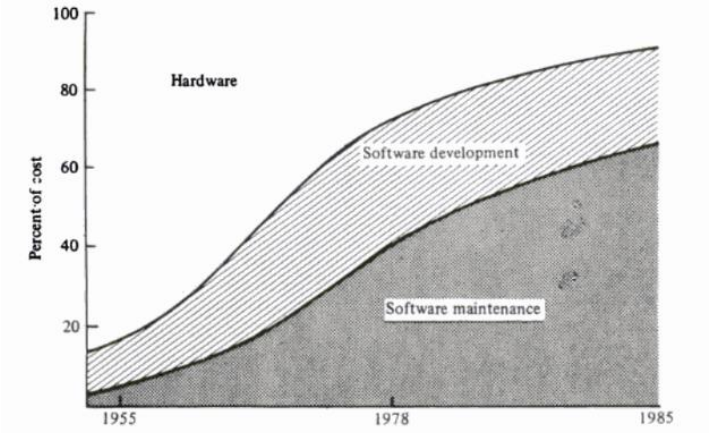


Figure 1.1 Changing hardware-software cost ratio *(from BOE76)*.

2. Distribution of effort in the software life cycle

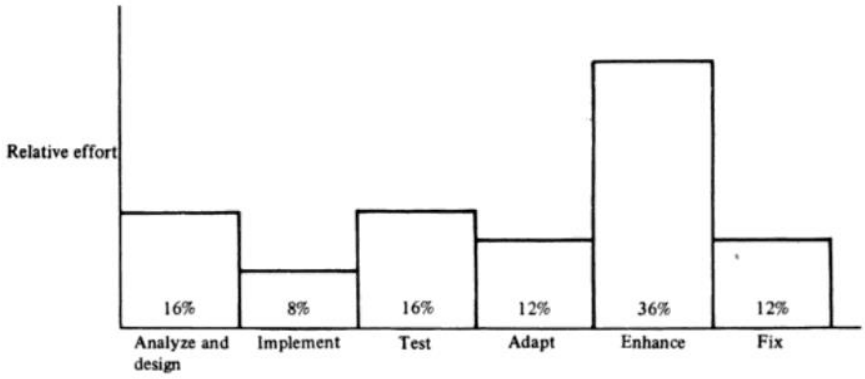


Figure 1.2 Distribution of effort in the software life cycle *(Boehm, Datamation 1973)*.

- Software maintenance activities consume more resources than software development activities.
- A large percentage of effort is devoted to software enhancement.
- Testing requires almost half the effort of software development.

Development	**Maintenance**
Period - 1-3 yrs	5-15 yrs.
Analyze & design, implement, test	adapt, enhance, fix
16% + 8% + 16%= 40%	12% + 36% + 12%= 60%
Other Ratios of Development/ Maintenance	40/60 30/70 10/90

3. **Project size categories**

Table 1.1 Size categories for software products

Category	Number of programmers	Duration	Product size
Trivial	1	1–4 wks.	500 source lines
Small	1	1–6 mos.	1K–2K
Medium	2–5	1–2 yrs.	5K–50K
Large	5–20	2–3 yrs.	50K–100K
Very large	100–1000	4–5 yrs.	1M
Extremely large	2000–5000	5–10 yrs.	1M–10M

4. **How programmers spend their time**

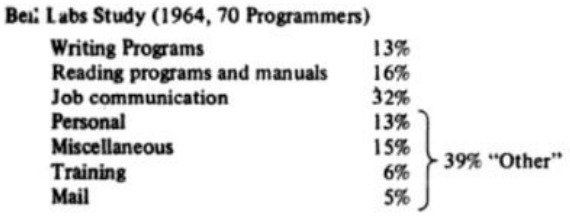


Figure 1.3 How programmers spend their time.

QUALITY AND PRODUCTIVITY FACTORS

Factor that influence quality and productivity

1. **Individual ability** – Productivity and quality are direct functions of individual ability and effort. There are two aspects to ability.:The general competence of the individual and familiarity of the individual with the application area.
   Lack of familiarity with the application area can result in low productivity and poor quality.
2. **Team communication** –Brooks had observed that the number of communication paths among programmers grows as n(n-1)/2, where n is the number of programmers in a project team.
   Brook's Law - Adding more programmers to a late project may make it later.
3. **Product Complexity** – Three levels of complexity are as follows:Application programs, utility programs, system level programs.
   - Application programs include scientific and data processing routines written in a high level language.
   - Utility programs include compilers, assemblers, linkage editors, loaders.
   - System programs includes data communication packages, OS routines.
4. **Appropriate notations** – The appropriate notations act as vehicles of communication among project personnel. Automated software tools may be used to verify proper usage.

5. **Systematic approaches** are not well developed or standardized. A single approach to software development and maintenance will not be adequate to cover all situations. So various approaches to software development should be used depending on the situation.
6. **Change control** – Requirement may change due to poor understanding of the problem on the part of the user or programmer. Use of appropriate notations and techniques makes controlled changes possible without degrading the quality of the work product. There must be a trade- off between cost and benefit of change.
7. **Level of technology** – In includes the factors such as programming languages, machine environment, programming practices, software tools.
8. **Required reliability** – More reliability results in decrease in productivity. Extreme reliability is gained only with great care in analysis, design, implementation, system testing and maintenance of software products. Increased level of human, hardware and software resources is required to achieve higher reliability.
9. **Available time** – Software projects are sensitive not only to total effort but also to elapsed time and the number of people involved.
   Utilizing 6 programmers for 1 month will be less effective than using 1 programmer for 6 months. On the other hand, using 2 programmers for 3 months may be more effective because of optimum level of communication paths and reinforcement. Determining optimum staffing levels and proper elapsed times for various activities in software product development is an important and difficult aspect of cost and resource estimation.
10. **Problem understanding -** Failure to understand the problem common and difficult issue. Customer does not understand the nature of the problem.
    Software engineer does not understand the application area and has trouble in communicating with the customer.
    Careful planning, customer interviews, task observation, prototyping, preliminary version of user's manual can increase both customer and developer understanding of the problem to be solved.
11. **Stability of requirements** – Change in requirements causes cost increase.
12. **Required skills** – Different kind of skills needed for different operations. In includes good communication skills, knowledge of the application area, problem solving skills and debugging skills.
13. **Facilities and resources** – Work related factors such as good machine access, quiet place of work will result in high programmer productivity and job satisfaction.
14. **Adequacy of training** – Boehm described the skills mostly needed in entry level programmers are team work, communication skill, technical skill and skill to develop and validate software requirement and design specifications.
15. **Management Skills** – Technical expertise is not accompanied by management skills. Managers find software project management to be difficult due to the differences in design methods, notation, and development tools.

16. **Appropriate goals** – The primary goal of software engineering is development of software products that are appropriate for their intended use. Every software product should provide optimum levels of generality, efficiency and reliability.
17. **Rising expectations** – There must be a trade-off between cost and benefit of satisfying rising expectations. There are 2 aspects. 1. How much functionality, reliability and performance can be provided by given amount of development effort. 2. Issue of fundamental limitations of software technology.
18. **Other factors** – Stability of computing system used to develop or modify the software.
    Memory and timing constraint of the software product.
    Experience of the programmer with the programming language.

Managerial Issues

SNo	Management problems
1	Poor project planning
2	Poor selection procedure to select project managers
3	Poor accountability regarding various project functions
4	Inaccurate estimate of resources
5	Inappropriate success criteria. Products are unreliable
6	Organizational structure selection – no proper decision rules
7	Management technique selection – no proper decision rules
8	Procedures, techniques, strategies to control project - not available
9	Procedures, techniques, strategies to find progress of project - not available
10	Quality of performance and production – no measuring techniques available

SNo	Solutions
1	Educate and train personnel of all levels
2	Enforce use of standards, procedures, documentation
3	Analyze data from prior projects to adapt effective methods
4	Define objectives in terms of quality
5	Define quality in terms of deliverables
6	Establish success priority criteria
7	Allow for contingencies
8	Develop accurate cost and schedule estimates accepted by all people involved
9	Select project managers with ability to manage software projects
10	Specific work assignment and job performance standards to software developers

PLANNING A SOFTWARE PRODUCT

The steps in planning a software project are

Defining the problem

1. Developing a statement that defines the problem and specifies the goals to be achieved. Develop a definitive statement of the problem to be solved. It includes the description of the present situation, problem constraint and statement of the goals to be achieved. The problem statement should be phrased in the customer's terminology.
2. Justify a computerized solution for the problem. In addition to being cost effective the computerized system must be socially and politically acceptable.
3. Identification and gathering of baseline data.Attention is focused on the roles to be played by the major subsystems of the computing system. A computing system consists of people subsystem, hardware subsystem, software subsystem, and interfaces among the subsystems.
4. Determining the system level goals and requirements. The function to be performed by each subsystem must be identified.
5. Establish high level acceptance criteria for the system

Developing a solution strategy.

6. Outlining solution strategies.Several solution strategies might have been chosen by the planners in order to perform feasibility studies and to prepare preliminary cost estimates.
7. The feasibility of each proposed solution strategy is determined.
    - The selected strategy provides a framework for design and implementation of the software product.
    - The best strategy is the combination of ideas from several different approaches and it may become apparent only after all the obvious solutions have been enumerated.
    - 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.
    - Techniques for determining the feasibility of a solution strategy includes case studies, worst case analysis and construction of prototypes.
8. When recommending a solution strategy it is important to document the reasons for rejecting other strategies.
9. A solution strategy should include a priority list of product features.

Planning the development process.

10. Define a life-cycle model and an organizational structure for the project.
11. Plan the configuration management, quality assurance and validation activities.
12. Determine phase-dependent tools, techniques, and notations to be used.
13. Establish preliminary cost estimates for system development.
14. Establish preliminary development schedule.
15. Establish preliminary staffing estimates.
16. Develop preliminary estimate for the computing resources required for the project.
17. Prepare a glossary of items.
18. Identify sources of information and refer them throughout the project plan.

Goals and requirements of software engineering

- After giving a concise definition we have to set goals and achievements that have to be achieved during the course of the project.
- The goals are targets for achievement and serve to establish a framework for a software development project.
- Apply to development process and work products.
- They may be 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 qualitative product goal: The system should reduce the cost of transaction by 25%.
- The other goals can be different types a few them are
  i. Every product should be useful, reliable, understandable and cost-effective.
  ii. Transportability, early delivery of subset capabilities.
  iii. Ease of use by non-programmers.
  iv. They may depend and vary on every new situation.
- **Requirements** specify capabilities that a system must provide order to solve a problem. They include
  i. Functional requirements
  ii. Performance requirements
  iii. Requirements for resources (hardware, firmware, software & user interfaces)
  iv. They must also specify development standards and quality assurance standards for both project and product.
- Quantitative requirements are those in which
  i. Phase accuracy shall have error not greater than 0.5 degree.
  ii. Response to extended interrupts shall be 0.25 seconds maximum.
  iii. System shall reside in 50k bytes of primary memory excluding file buffers.
  iv. System shall be operational 95% of each 24 hour period.
- Qualitative requirements are those with the following criteria
  i. Accuracy shall be sufficient to support mission.
  ii. System shall provide real-time response.
  iii. System shall make efficient use of primary memory.
  iv. System shall be 99% feasible.
- High level goals and requirements can often be expressed in terms of quality attributes a system should posses. High-level attributes can be expressed in terms of attributes that can be built in to work products. Eg. reliability can be expressed in terms of source code accuracy, robustness, completeness and consistency.

Define the quality attributes

1. **Portability** – The ease with which software can be transferred from one computer system or environment to another.
2. **Reliability** – The ability of a program to perform a required function under stated conditions for a stated period of time.
3. **Efficiency** – The extent to which software performs its intended function with a minimum consumption of computing resources.
4. **Accuracy** – (1) A qualitative assessment of freedom from error (2) A quantitative measure of the magnitude of error, expressed as a function of the relative error.
5. **Error** – A discrepancy between a computed value or condition and the true, specified, or theoretically correct value or condition.
6. **Robustness** – The extent to which software can continue to operate correctly despite the introduction of invalid inputs.
7. **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 user expectations.

PLANNING THE DEVELOPMENT PROCESS

1. **Phased Life Cycle Model**
2. **Milestones, Documents, and reviews**
3. **Cost Model**
4. **Prototype Life Cycle Model**
5. **Successive Versions**

PHASED LIFE CYCLE MODEL - The phased model segments the software life cycle into a series of successive activities. Each phase requires well defined input information, well defined processes and results in well defined products.

The phased model consists of the following phases: analysis, design, implementation, system testing and maintenance. This model is also called water fall model.

Analysis phase consists of 2 sub phases 1. Planning 2. Requirements Definition

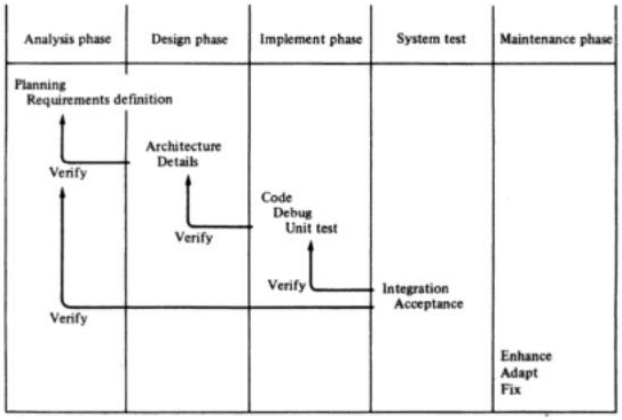


Figure 2.2 The phased model of the software life cycle.

1. Planning Activities during planning are

- Understanding the customer's problem
- Performing feasibility study
- Developing a recommended solution strategy
- Determining the acceptance criteria
- Planning the development process

The products of planning are System Definition and Project Plan.

The System Definition is expressed in English or some other natural language and may include charts figures, tables and equations.

The Project Plancontains the life cycle model to be used , the organizational structure for the project, preliminary development schedule, preliminary cost and resource estimates, preliminary staffing requirements, tools, techniques, standard practices to be followed.

2. Requirements Definitionis concerned with identifying the basic functions of software component in a hardware/software/people system.

Design phaseis concerned with identifying software components, specifying relationships among components, providing a blue print for the implementation phase. Design consists of Architectural Design and Detailed Design.

Architectural Designinvolves identifying the software components and specifying interconnections among the components.

Detailed Designinvolves adaptation of existing code, modification of standard algorithms, invention of new algorithms and packaging of the software product.

Implementation phaseinvolves translation of design specification into source code, debugging, documentation and unit testing of source code.

System testing phase involves two kinds of activities namely, integration testing and acceptance testing.

Maintenance phaseincludes enhancement of capabilities, adaptation of the software to the new processing environments and correction of software bugs.

MILESTONES, DOCUMENTS, AND REVIEWS

Establishing milestones, review points, standardized documents and management sign-offs can improve product visibility. The development process becomes a more public activity and the product becomes more tangible.

The following activities are performed.

1) System Definition and Project Plan are prepared. A product feasibility review is held to determine the feasibility of project continuation.
2) The preliminary version of User's Manual provides a vehicle of communication between customer and developer.
3) Software Requirements Specification clearly and precisely defines each essential requirement of the software product.
4) A preliminary version of Software verification Plan states methods to be used and the results to be obtained in verifying each of the requirements stated.

System Definition includes

1. Problem Definitions, System justification, Goals, constraints
2. Functions (hardware, software, people)
3. User characteristics, Development environment
4. Solution Strategy
5. Acceptance Criteria

Reviews and milestones in the phased life-cycle model.

REVIEW	**WORK PRODUCTS REVIEWED**
PFR - Product Feasibility Review	System Definition, Project Plan
SRR – Software Requirements Review	Software Requirements Specification, Preliminary User's Manual, Preliminary Verification Plan
PDR – Preliminary Design Review	Architectural Design Document
CDR - Critical Design Review	Detailed Design Specification, User's Manual, Software Verification Plan
SCR – Source Code Review	Walkthrough & Inspections of the Source Code
ATP – Acceptance Test Review	Acceptance Test Plan
Product Release Review	All of the above
PPM Project Post-Mortem	Project Legacy

Project Plan includes

1. Life Cycle Model
2. Organizational structure
3. Resources, Schedule, Estimate
4. Project Control mechanisms
5. Tools, techniques, languages
6. Testing, maintenance
7. Mode of delivery

Outline of User's Manual

1. Product overview
2. Help mode
3. Modes of operation

Software Requirements Specification includes

1. Product overview
2. Development/ operational/maintenance environment
3. External interfaces
4. Performance requirement
5. Exception handling
6. Acceptance criteria

COST MODEL OF THE SOFTWARE LIFE CYCLE

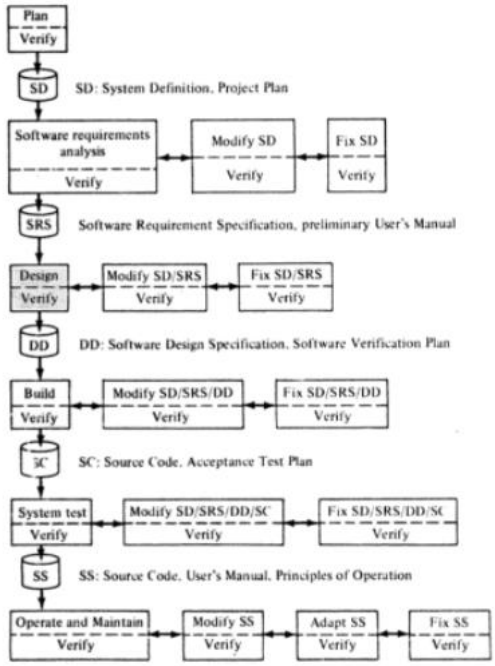


Figure 2.4 The cost model of the software life cycle *(ALF82)*.

The cost of conducting a software project is the sum of costs incurred in conducting each phase of the project.

1. The cost of producing the System Definition and Project Plan
2. The cost of preparing Software Requirements Specification plus the cost of modifying and correcting System Definition and Project Plan
3. Cost of Design is the cost of software design specification and Software Verification Plan
4. The cost of System testing includes cost of planning and conducting tests.
5. The cost of software maintenance is the sum of cost of performing product enhancements, adaptations to new requirements and fixing bugs.

Modifications or corrections to the SRS in subsequent phases of the life cycle are so costly.

Not only the document has to be modified, but all the intermediate work products must be updated.

- This type of software life cycle is obtained by considering the cost of performing the various activities in a software project. The cost of conducting a software project is the sum of the costs incurred in the conducting each phase of the project.
- Costs incurred within each of the phase include the cost of performing the processes and preparing the products for the phase, plus the cost of verifying that the products of the present phase are complete and consistent with respect to all previous phases.
- Modifications and corrections to the products of previous phases 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.
- For example, the cost of producing the system definition and the project plan is the cost of performing the planning function and preparation the documents, plus the cost of verifying that the system definition accurately states the customer's needs and the cost of verifying that the project plan is feasible.

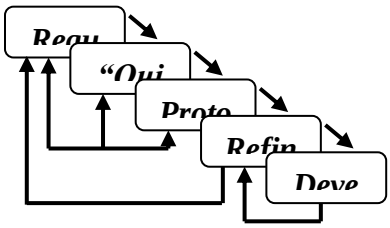
PROTOTYPE LIFE CYCLE MODEL

A prototype is a mock-up or model of a software product.There are several reasons for developing a prototype. The reasons are:

1. It is used to illustrate input data formats, messages, reports, and interactive dialogues for the customer and for gaining better understanding of the customer's needs.
2. It is used to explore technical issues in the proposed product.
3. It is developed in a situation the phased model is not appropriate.

Development of a totally new product will involve prototyping during the planning and analysis phase or the product may be developed by iterating through a series of successive designs and implementations. This technique is referred to as the method of successive versions.

- Prototyping is a process that enables the developer to create a model of the software to be built
- Often the customer will defines a set of general objectives for software, but will not identify detailed input, processing, or output requirements. In other cases the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the human-machine interaction should take. In these and many other situations the prototyping approach to software engineering may be best.



- Prototyping is a process that enables the developer to create a model of the software to be built.
- The model can take one of the three forms; a paper prototype that depicts human-machine interaction in a form that enables the user to understand how such interaction will occur, a working prototype that implements some subset of the function required the desired software, or an existing program that perform part or all of the function desired but has other features to be improved upon in the new development effort.
- In most projects, the first system is barely usable. It may be too slow, too big, awkward in use, or all three. There is no alternative but to start again, but smarter, and build a redesigned version in which these problems are solved.
- When a new system concept or new technology is used, one has to build a system to throw away, for even the best planning it is impossible to get it right the first time.
- Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or existing tools that enable working program to be generated quickly.

SUCCESSIVE VERSIONS

Product development by the method of successive versions is an extension of prototyping in which an initial product skeleton is refined into increased levels of capability.

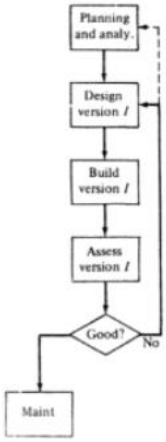


Figure 2.7a Design & implementation of successive versions.

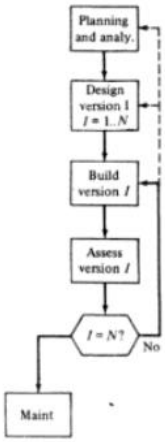


Figure 2.7b Analysis & design followed by implementation of successive versions.

The characteristics of each successive design will have been planned during the analysis phase. The dashed lines in the figure indicate that implementation of the I^{th} version may reveal the need for further analysis and design before proceeding with implementation of Version I+1.

PLANNING AN ORGANIZATIONAL STRUCTURE

A proper organization structure is most important as it helps in decision making and the tackling of any problem in the best possible way. Now the planning of the organizational structure can be explained in the following ways.

1. PROJECT STRUCTURE

There are three approaches in the formation of project structure

- **Project Format**

  This is a format where a team of programmers is grouped and the look in to all the steps in the development of a project. This may range from 1 to 3 years. They will be shifted to new projects only on the completion of the project. Some of the team members stay back to look into the maintenance of the finished project.

- **Functional Format**

  In the functional approach to organization, a different team of programmers performs each phase of the project, and the work products pass from one team to the other. A typical variation on the functional format involves three teams: an analysis team, a design and implementation team, and a testing and maintenance team.

  Team members are periodically transferred from function to function to avoid overspecialization. This requires more communication between teams but personnel become specialists in particular roles and results in more attention to proper documentation.

- **Matrix Format**

  In this organization each of the function has its own management team and a group of specialist personnel who are concerned only with that function. In matrix organizations, everyone has at least two bosses, and the need to resolve ambiguity and conflict is the price paid for project accountability.

2. PROGRAMMING TEAM STRUCTURE

Every programming team must have an internal structure. The best team structure for any particular project depends on the nature of the project and the product.A large project may utilize several programming teams. In this case, each team should have responsibility for a well defined functional unit that communicates with the other units through well-defined interfaces.

Democratic teams

These are termed as “ego-less team” which has one team leader who assumes the position of the first among the equals. The advantages of this structure are each member contributes to decision-making; knowledge exchange between team members is smooth increased job satisfaction. The disadvantages are communication overheads which lead to delayed decisions.

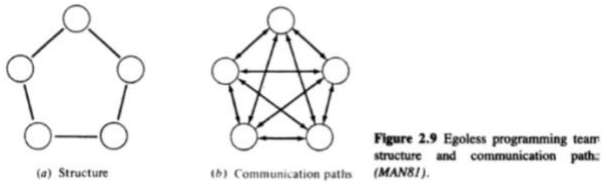


Figure 2.9 Egoless programming team structure and communication paths *(MAN81)*.

Chief programmer teams

They are highly structured and efficient allocations done by the chief programmer. There is the presence of the librarian who maintains the documentation. The emphasis in a chief programmer structure is to provide a complete technical and administrative support to the chief programmer, who has responsibility and authority for development of the software product. The advantages are centralized decision making and reduced communication paths. This can also result low morale for sub-ordinate programmers

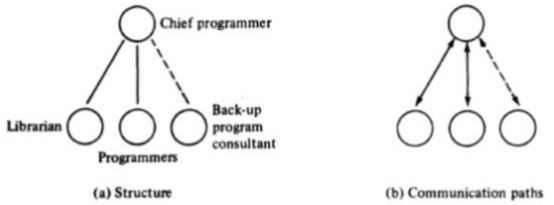


Figure 2.10 Chief programmer team structure and communication paths *(MAN81)*.

.**Hierarchical team structure**

This is a middle organizational structure between the previous two. This limits the number of communication paths

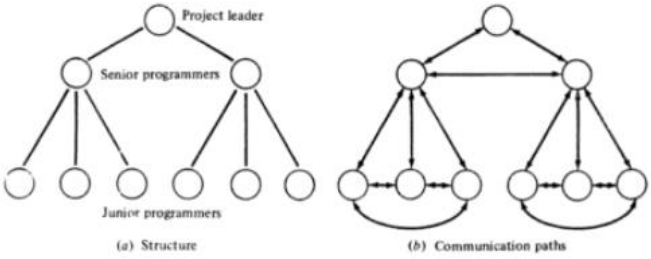


Figure 2.11 Hierarchical programming team structure and communication paths *(MAN81)*.

This will be a very effective structure when a large project is being handled and many subsystems are working over it. Hierarchical teams are particularly well suited to development of hierarchical software products, because each major subsystem in the hierarchy can be assigned to a different programming team.

The major disadvantage is that technically competent programmers may be promoted into management positions. This may have two different effects based on the personality traits of the respective person being promoted.

3. MANAGEMENT BY OBJECTIVES

Using MBO, employees set their own goals and objectives with the help of their supervisor, participate in the setting of their supervisor's goals, and are evaluated by meeting their objectives. They are mostly set for periods of 1 to 3 months.

SOFTWARE ENGINEERING
UNIT II

INTRODUCTION:

Estimating the cost of a software product is one of the difficult and error-prone tasks. Due to the difficulty, some organizations use a series of cost estimates. Each estimate is a refinement of the previous one, and is based on the additional information gained as a result of additional work activities.

- most difficult task in software engineering
- difficult to make estimate during planning phase
- series of cost estimation
- preliminary estimate is prepared during planning
- an improved estimate is presented at the software requirements review
- final estimate is prepares at the preliminary design view

1.1 SOFTWARE COST FACTORS:

The major factors that influence the cost of a software product are:

1. Programmer Ability
2. Product Complexity
3. Product Size
4. Available Time
5. Required Level of Reliability
6. Level of Technology

i. Programmer Ability :

- The goal was to determine the relative influence of batch and time-shared access on programmer productivity.
- The resulting differences from an experiment conducted by Harold Sackman with 12 experienced programmers were given two problems to solve some using batch facilities and some using time-sharing. The resulting differences in individual performance among the programmers were much greater than could be attributed to relatively small effect of batch or time-shared machine access.
- The differences between the worst and best were factors of 6 to 1 in program size, 8 to 1 in execution time, 9 to 1 in development time, 18 to 1 in coding time and 28 to 1 in debugging time. Sackman observed a productivity variation of 16 to 1.
- On very large projects, the differences in individual programmer ability will tend to average out, but in small individual differences in ability can be significant.

ii. Product Complexity :

There are three acknowledged categories of software products:

1. Application Programs (Data Processing and Scientific)
2. Utility programs (Compilers, Linkage editors, and inventory Programs)
3. System Level Programs (DBMS, OS, Real-time)

(a) Application Programs:

□□□ □Application programs include data processing and scientific programs. Application programs are developed in environment provided by the language compilers such as FORTRAN, PASCAL

Application Programs:

PM=2.4*(KDSI)1.05**

TDEV=2.5*(PM)0.38**

where PM = Effort in Programmer Months

KDSI =No. of thousands of delivered source instructions

TDEV = Development Time

MO= Months

(b) Utility Programs:

Utility programs include compilers, assemblers. □Utility programs are written to provide user processing environment

Utility Programs:

PM=3.0*(KDSI)1.12**

TDEV=2.5*(PM)0.35**

(c) System Level Programs:

System programs include operating system, DBMS, real time system. System programs interact directly with the hardware

System Programs:

PM=3.6*(KDSI)1.2**

TDEV=2.5*(PM)0.32**

Brook's states that utility programs are three times as difficult to write as application programs and system programs are three times as difficult to write as utility programs. Product complexity are 1-3-9 for application, utility, system programs. Boehm uses three levels of product complexity equations of total programmer month of effort pm is provided in terms of the number of thousands of delivered source instruction KDSI,

programmer cost for the software project = the effort in programmer month*cost per programmer month

In this terminology the three levels of product complexity are organic, semidetached, embedded.

They roughly correspond to organic-application, utility-semidetached, embedded-system.

application program:pm=2.4*(KDSI)**1.05

utility programs:pm=3.2*(KDSI)**1.12

system programs:pm=3.6*(KDSI)**1.20

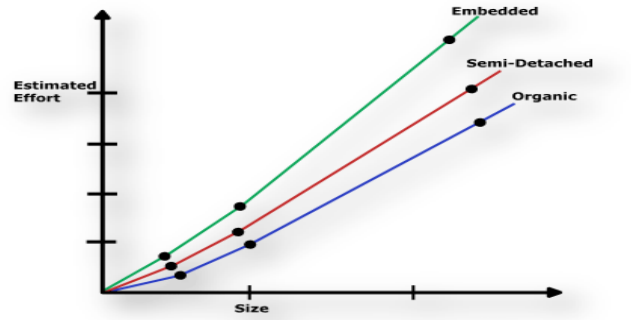


Fig. Effort Verses Product size

xample: For a development of a 60,000 line application programs, utility programs and system programs the ratio of pm:1 to 1.7 to 2.8.

The development time for a program
application program TDEV=2.5*(pm)**0.38
utility programs TDEV=2.5*(pm)**0.35
system programs TDEV=2.5*(pm)**0.

The total programmer months for a project and the development time the average staffing level is obtained by

Applicationprogram: 176.6pm/17.85mo=9.9programmers
utility program:294pm/18.3mo=16programmers
system program:489.6pm/18.1mo=27programmers

Failures in estimating the numberof source instructions in a software product is due to under-estimating the amount of house-keeping code required.

HOUSE KEEPING CODE

Portion of the source code that handles input,output,interactive user communication,error checking and error handling.

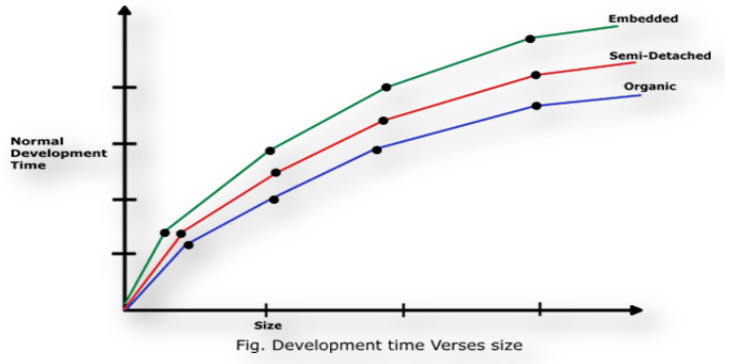


Fig. Development time Verses size

(iii) Product Size:

- A large software product is more expensive to develop than a small one.
- Boehm equation indicate that the rate of increase in required effort grows with number of source instruction at an exponential

- Using exponents of 0.91 and 1.83 results in estimatesof 1.88 and 3.5 more effort for a product that is twice as large ,and factors of 8.1 and 67.6 for products that are 10 times as large as known product
- These estimates differ by factors of 1.86(3.5/1.88)for products that are twice as large and 8.3(76.6/8.1) for products that are 10 times as large

Effort equation	Schedule equation	Reference
PM=5.2(KDSI)**0.91	TDEV=2.47(MM)**0.35	(WAL77)
PM=4.9(KDSI)**0.98	TDEV=3.04(MM)**0.36	(NEL78)
PM=1.5(KDSI)**1.02	TDEV=4.38(MM)**0.25	(FRE79)
PM=2.4(KDSI)**1.05	TDEV=2.50(MM)**0.38	(BOE81)
PM=3.0(KDSI)**1.12	TDEV=2.50(MM)**0.35	(BOE81)
PM=3.6(KDSI)**1.40	TDEV=2.50(MM)**0.32	(BOE81)
PM=1.0(KDSI)**1.50	-	(JON77)
PM=0.7(KDSI)**1.50	-	(HAL77)

- Depending on the exponent used we can easily be off by a factor of 2 in estimating effort for a product twice the size of a known product and by a factor of 10 for a product 10 times the size of known product,even if all other factors that influence cost remain constant.

(iv) AVAILABLE TIME

- Total project effort is sensitive to the calendar time available for project competition
- Software projects require more total effort,if development time is compressed or expanded from the optional time
- According to Putnam, project effort is inversely proportional to the fourth power of the development time E=k/(TD**4)
- This formula predicts zero effort for infinite development time
- Putnam also states that the development schedule cannot be compressed below about 86%of the nominal schedule regardless of the number of people or resources utilized
- Boehm states that “there is a limit beyond which a software project cannot reduce its schedule by buying more personnel and equipment “

(v)REQUIRED LEVEL OF RELIABILITY

The ability of a program to perform a requiredfunction under stated conditions for a stated period oftime

- Accuracy
- Robustness
- Completeness
- Consistency

- These characteristics can be built in to a software product
- There is a cost associated with different phases to ensure high reliability
- Product failure may cause slightly inconvenience to the user
- While failure of other products may incur high financial loss or risk to human life

Development effort multiliers for software reliability

Category	Effect of failure	Effort multiplier
Very low	Slight inconvinience	0.75
Low	Losses easily recovered	0.88
Nominal	Moderately difficult to recover loses	1.00
High	High financial loss	1.15
Very high	Resk to human life	1.40

(vi) LEVEL OF TECHNOLOGY

In a software development required project is reflected by

1. programming language
2. abstract machine
3. programmingpractices
4. software tools used

odern programming languages provides additional features to improve programmer productivity and software reliability.These features include

1. strong type checking
2. data abstraction
3. separate computation
4. exception handling

- productivity will suffer if programmers must learn a new machine environment as part of the development process
- modern programming practices include the use of
  a) systematic analysis and design technique
  b) structure designed notations
  c) inspection
  d) structured coding
  e) systematic testing
  f) program development library
- software tools range from elementary tools such as assemblers compilers,interactive text editors and DBMS.

1.2 SOFTWARE COST ESTIMATION TECHNIQUES :

- software cost estimates are based on past performance
- cost estimates can be made either (i)top down (ii)bottomup
- Top-down:focus on system level cost such as computer resources, personnel level required to develop the system
- Bottom up:The costs to develop each module or subsystem are combined to arrive at an overall estimate.

Software Cost Estimation Techniques are :-

1. Expert Judgment

2. Delphi Cost Estimation

3. Work Breakdown structures

4. Algorithmic models (COCOMO)

1. Expert Judgment :

Most widely used cost estimation techniques(top down)

Expert judgment relies on the experience background and business sense of one or more key people in an organization.

Eg: an expert might arrive at a cost estimate in the following manner.

1. To develop a process control system
2. It is similar to one that was developed last yr in 10 months at a cost of one million dollar
3. It was not a respectable profit
4. The new system has same control functions as the previous but 25% more control activity
5. So the time and cost is increased by 25%
6. The previous system developed was the same
7. Same computer and controlling devices and many of the same people are available to develop the new system therefore 20% of the estimate is reduced
8. Resume of the low level code from the previous reduces the time and cost estimates by 25%
9. This results in estimation of eight lakhs $ and eight months.
10. Small margin of safety so eight lakhs 50,000$ and nine months development time
11. Advantage: experience

So the net effect will be reduced to 20% of reduction results in $80,000 and 1 year delivery time.

Advantages:-

Experience can be a liability but may have overlooked some factors that make the new project significantly different.

Disadvantages:-

The major disadvantage of group estimation is the effect that interpersonal group dynamics may have on individuals in the group.

2. Delphi Cost Estimation:

□□□□□□□This technique was developed at the Rand Corporation in 1948

- In Delphi Technique Estimators estimates the System Definition document given by coordinator without discussing each other.
- This process is iterated until the co-coordinator received the correct estimation from the estimators.
- In Delphi cost estimation the same process is followed except the estimators prepares their statements anonymously after a discussion.

- This process is iterated until the co-coordinator received the correct estimation from the estimators.

This technique can be adapted to software estimation in the following manner

1. A coordinator was selected and the System Definition was provided.
2. Estimators study the document and complete their estimates
3. They ask questions to the coordinator but they won't discuss with one another
4. The co-coordinator prepares and distributes a summary of the estimators response
5. The estimators complete another estimate from the previous estimator
6. The process is iterated for as many as required

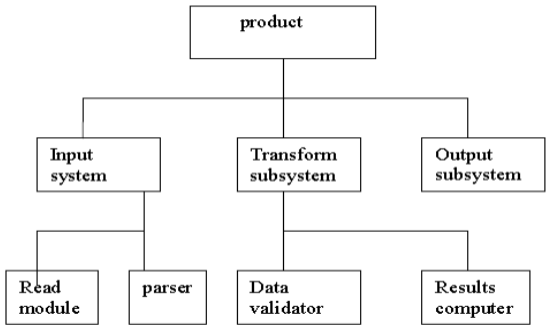
3. Work Break Down Structure:

- ✓ A bottom-up estimation tool
- ✓ WBS is a hierarchical chart that accounts for the individual parts of the system
- ✓ WBS chart can indicate either product hierarchy or process hierarchy

PRODUCT HIERARCHY

- ✓ It identifies the product components and indicates the manner in which the components are interconnected.

Fig. The product work break down structure(WBS)



PROCESS HIERARCHY

- ✓ It identifies the work activity and relationship among those activities
- ✓ Using WBS cost are estimated by assigning cost toeach individual component in the chart and summing the cost
- ✓ WBS are the most widely used cost estimation techniques

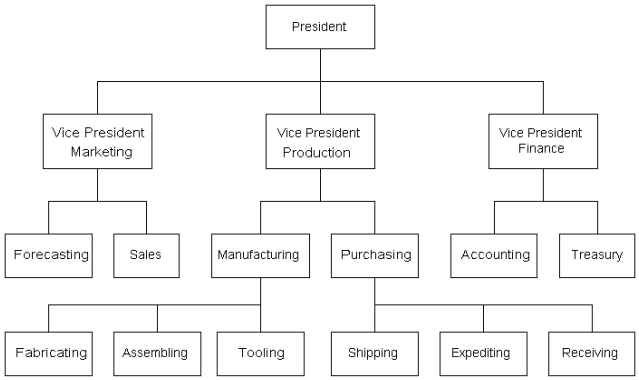


Fig. The Process work break down structure (WBS)- Manufacturing industry

The process work break down structure analyses the break-up of the different processes involved in a project, like planning, design, development, system test and modify. Each has many subsystems showing sub-processes. Some planners use both product and process work break down structure charts for cost estimation.

(d) Algorithmic Cost Model :

Constructive Cost Model (COCOMO)

Algorithmic cost estimators compute the estimated cost of software system as the some of the cost of the module this model is bottom up estimates. The constructive cost model (COCOMO)is an algorithmic cost model described by Boehm

In COCOMO model the equation calculates the programmer month and development schedule are used for the program unit based on the number of delivered source instruction(DSI). Effort multipliers are used to adjust the estimate for product attribute, project attributes, computer attributes, and personnel attributes.

The effort multipliers examines the data from 63 project and by using Delphi technique The COCOMO equation incorporates a number of assumptions.

The organic mode application program equation applied in the following types ofsituations.

- small to medium size project
- familiar application area
- stable well understood virtual machine
- in house development effort

Effort multipliers are used to modify these assumptions

- It includes cost of documentation and review
- It includes cost of program managers and program librarian
- Covers design through acceptance

Software project estimated by COCOMO model include the following:

- The requirements remain the same throughput the project
- Definition and validation techniques of n architecture design is performed by a small number of capable people
- Detailed design, coding and unit testing are performed in similar by a group of programmers working in teams.

- ✍ Integration testing is based on early test planning.
- ✍ Interface errors are mostly found by unit testing and by walkthroughs and inspections.
- ✍ Documentation is performed incrementally as a part of the development process.

COCOMO Effort Multiplier

Multiplier	Range of values
Product attributes:	
Required reliability	0.75to1.40
Database size	0.94to1.16
Product complexity	0.70to1.65
Computer attributes:	
Execution time constraint	1.00to1.66
Main storage constraint	1.00to1.56
Virtual machine volatility	0.87to1.30
Computer turnaround time	0.87to1.15
Personnel attributes:	
Analyst capability	1.46to0.71
Programmer capability	1.42to0.70
Applications experience	1.29to0.82
Virtual machine experience	1.21to0.90
Programming language experience	1.14to0.95
Project attributes	
Use of modern programming practises	1.24to0.82
Use of software tools	1.24to0.83
Required development schedule	1.23to1.10

Example

- A 10 KDSI software product for telecommunications processing.
- PM=2.8*(10)**1.20=44.4
- TDEV=2.5*(44)**0.32=8.4
- Effort adjustment factor of 1.17 is used and the estimation is 51.9 PM and 8.8 TDEV.
- Dollars=(51.9 PM)*($6000 per PM)=$311,400
- Now the adjustment factor 1.58 and the amount becomes $350,760

Conclusion

- COCOMO can be used to gain insight into the cost factors within an organization.
- The weakness of COCOMO is that use of multiplicative effort adjustment factor assumes that the various effort multipliers are independent.

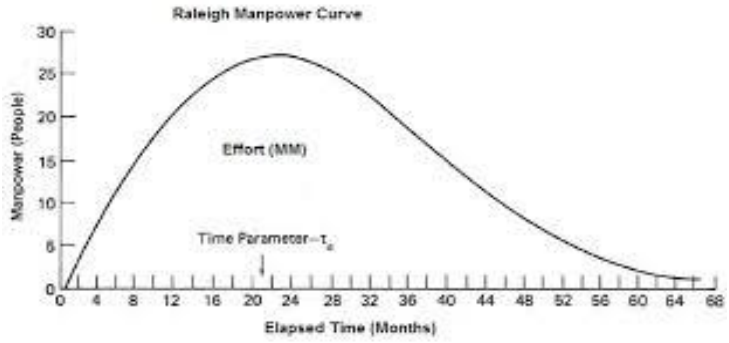
1.3 STAFFING LEVEL ESTIMATION

- the number of personnel required throughput a software development project is not constant
- planning an analysis are performed by a small group of people □architectural design by a larger or smaller group detailed design by a larger number of people
- Implementation and system testing requires the largest number of people
- in 1958 Norden observed that research and development project follows a cycle of planning,design,prototype development and use with corresponding personnel utilization.

RAYLEIGH EQUATION:

- Any particular point on the Rayleigh curve represents the number of fulltime equivalent personnel required at the instant in time
- Rayleigh curve is specified by two parameters
  - td-the time at which the curve reaches its maximum value
  - k-the total area under the curve. That is the total effort required for the project
- In 1976 Putnam studied 50 army software life cycle using Rayleigh curve
- From his observation , Rayleigh curve reaches its maximum value td,during system testing and product release for many software products

- $FSP=\bar{P}M\frac{(0.15TDEV+0.7t)}{(0.25(TDEV)^2} - \frac{(0.15TDEV+0.7t)^2}{0.15(TDEV)^2}$



Putnam's interpretation of the Rayleigh's curve is illustrated in the fig. The planning, requirements analysis, and functional design (external and architectural design) are not included in the project curve.

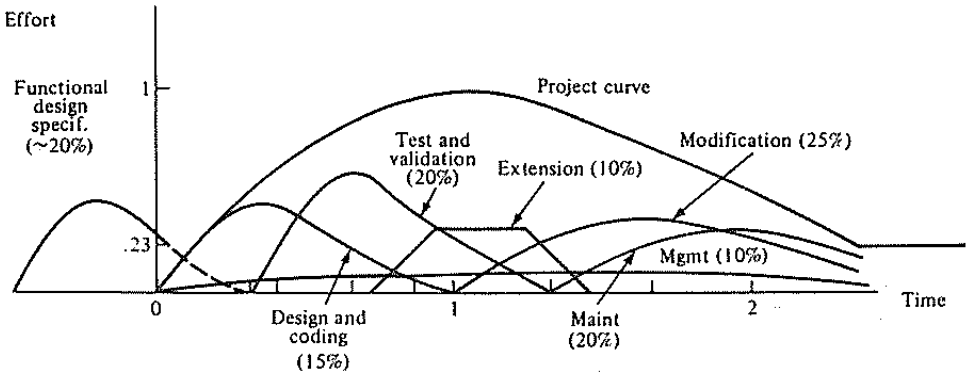


Figure 3.8 Putnam's interpretation of the Rayleigh Curve *(PUT76)*.

From Boehm's observation:Rayleigh curve is an accurate estimator of personal requirements for the development cycle from architectural design through implementation and system testing if the portion of the curve between 0.3td and 1.7td is used. The Rayleigh curve is then in the form

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

ESTIMATING SOFTWARE MAINTENANCE COST:

- Software maintenance cost requires 40 to 60%of the total life cycle devoted to software product
- In some cases it may be 90%
- Maintenance activities include

  1. enhancement to the product
  2. adapting the product to new processing environment
  3. correcting problems

Distribution and maintenance activities includesenhancement-60%, adaptation-20%,error correction- 20%.

During planning phase of the software project the major concerned about the maintenance are

(i) estimating the number of maintenance programmers that will be needed

(ii) specifying the facilities required for maintenance

A widely used estimators of maintenance personnel is the number of source lines that can be maintained by an individual programmers

Software Requirements Specification:

Format of a Software Requirements Specification:

Section 1: Product overview and summary

Section 2: Development, Operating and Maintenance environment

Sections 1 & 2 presents and overview of product features and processing environments for development, operations and maintenance of the product.

Section 3: External Interfaces and Dataflow

It includes user displays, report formats, summary of user commands and report options, Data Flow Diagram and Data Dictionary.

Data flow diagrams: It specifies

1. Data sources
2. Data sinks
3. Data stores
4. Transformations to be performed on the data.
5. Flow of data between sources, sinks, transformations and stores.

   Informal DFD

   Formal DFD

   - Like flowcharts, DFD can be used at any level of detail.
   - DFD are not concerned with algorithmic details or decision structure.

A Data Dictionary Entry

Name : Create

Where used : DLP

Purpose : Create passes a user created design entity to the DLP processor for verification of syntax.

Derived from: User Interface Processor
Sub Items :

- Name
- Uses
- Procedures
- References

Notes: Create contains one complete user-created design entity.

Data Dictionary entry:

Entries in a data dictionary include the name of the data item, and attribute such as data flow diagrams, where it is used, its purpose, where it is derived from, its sub items and any notes that may be appropriate. Each data item on each DFD should appear in the data dictionary.

Section 4: Functional Requirements. This section specifies the functional requirements for the software product.

Functional requirements are expressed in relational state oriented notations that specify relationships among inputs, actions and outputs.

Section 5: Performance Requirements. Performance characteristics are as follows.

- Response time for various activities
- Processing time for various processes
- Throughput
- Primary and secondary memory constraints
- Telecommunication bandwidth

Performance characteristics must be stated in verifiable terms and methods to be used in verifying performance must also be specific.

Section 6: Exception handling. Exception handling includes the actions to be taken and messages to be displayed in response to undesired situations or events. A table of exception conditions and exception responses should be prepared. Possible exceptions include:

- Division by zero
- Temporary resource failure
- Permanent resource failure
- Incorrect, inconsistent or out of range input data.

Section 7: Early subsets and implementation priorities.

- Software products are sometimes developed as a series of successive versions.
- The initial version may be a skeletal prototype, that demonstrates basic user functions and provide a framework for the evolution of the product.
- Each successive version may include the functions of previous versions and provide additional processing functions
- Successive versions can provide increasing levels of capabilities.

Section 8: Foreseeable modifications and enhancements. If the designers and implementers are aware of likely changes, they can design and build the result in a manner that will ease those changes.

Section 9: Acceptance criteria. They specify:

- Functional and performance test that must be performed.
- Standards to be applied to source code.
- Internal documentation.
- External documents such as user's manual, installation and maintenance procedures.

Section 10: Design hints and guidelines : Requirements specification is concerned with functional and performance aspects of a software product, specifying product characteristics without implying how the product will provide those characteristics.

Certain understandings gained during planning and requirements definition should be recorded as hints and guidelines to the product designers.

Section 11: Cross reference index. It is used to index specific paragraph numbers in SRSto specific paragraphs in System Definition and Preliminary User's Manual and to other sources of information.

Section 12: Glossary of Terms. This section provides definitions of terms that may be unfamiliar to the customer and the product developer.

Desirable properties:

A requirements document should be

- Correct
- Complete
- Consistent
- Unambiguous
- Functional
- Verifiable
- Traceable
- Easily changed
  - Incorrect or incomplete set of requirements can result in a software product that satisfy its requirements, but does not satisfy customer needs.
  - Ambiguous requirement is subject to different interpretations by different people.
  - Software requirements should be functional in nature. They should describe what is required without implying how the system will meet its requirements.
  - Requirements must be verifiable from two points of view: it must be possible to verify that the requirements satisfy the customer's needs, and subsequent work products satisfy the requirements.
  - Requirements should be traceable to specific customer statements and to specific statement in the System Definition.
  - Changes will occur and project's success often depends on the ability to incorporate change without starting over.

Specification Techniques :

Specifying the functional characteristics of a software product is one of the most important activities to be accomplished during the requirement analysis.

- The formal notations are concise and unambiguous.
- They provide the basis for verification of the resulting software product.
- Two nations are used to specify the functional characteristics.
    1. Relational
    2. State oriented

Relational notations:

- It is based on the concept 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 named entities
Attributes specify permitted operations on entities, relationships among entities and data flow between entities.

Relational notations include,

- Implicit equations
- Recurrence relations
- Algebraic axioms
- Regular expressions

(i)Implicit Equations :-

- ✓ It states the properties of a solution without stating a solution method.
    E.g. Matrix inversion M*M^1=I+-E
- ✓ Matrix inversion is specified such that matrix product of the original matrix M and the inverse of M, M yields the identity matrix I± the error matrix e (computation error).
- ✓ Complete specification of matrix inversion must include items such as matrix size, type of data elements and degree of sparseness.
- ✓ Given a complete functional specification for matrix inversion, design involves specifying a data structure, an algorithm for computing the inverse and a packaging technique for the inverse.

(ii) Recurrence relations:-

- ✓ It consists of on initial part called the basis and one or more recursive parts.
- ✓ The recursive part describe the decide value of a function in terms of other value of the function

Eg. Fibonacci number, F1(0)=0,
F1(1)=1,

F1(N)=F1(N-1) belongs to
F1(N*2) for all N>1.

- ✓ Recurrence relation is easily transformed into recursive programs.
  Eg: Acker's function is defined as:-

A(0,K)=K+1.	for all K>0
A(J,0)= A(J-1,1)	for all J>0
A(j,k)=A(J-1,A(J,K-1))	for all j,k>0

(iii) Algebraic Axioms:

- ✓ The mathematical systems are defined by axioms .
- ✓ The axioms specify fundamental properties of a system and provide a basis for deriving additional properties that are implied by the axioms .
- ✓ These additional properties are called *theorems* .
- ✓ In order to establish a valid mathematical system, the set of axioms must be completed and consistent.ie.. it must be possible to prove desired resulting using the axioms,and it must not to be possible to prove contradictory result.
- ✓ Elegance ofdefinition is achieved if the axioms are independent (no axioms can be derived from the other axioms) . But in practice independence of axioms is less important than completeness and consistency.

The axiomatic approach :

- ✓ The axiomatic approach can be used to specify functional properties of software system.
- ✓ This approach can used to specify abstract datatype .
- ✓ A data type is characterized as a set of objects and a set of permissible operations on the objects.
- ✓ The term "*abstract data type*" (or) "*data abstraction*"are refers to the fact that permissible operations on the data objects are emphasized while representation details ofthe data object are suppressed.
- ✓ Axiomatic specification of the last-in-first out (LIFO)property ofstack objects is specified
- ✓ Operations NEW, PUSH,POP,TOP,EMPTY are named to suggest their purpose ,but the definition is not dependent on the particular names choosen.

(i) NEW – Creates a new stack
(ii) PUSH - Add a new items to the top of the stack
(iii) TOP - Returns a copy of the top items
(iv) EMPTY - Test for empty stack.

Note: Operation NEW yield a newly created stack. PUSH requires two arguments .

SYNTAX :

OPERATION	DOMAIN	RANGE
NEW	() -->STACK	
PUSH	(STACK ,ITEM)-->	STACK
POP	(STACK)-->	STACK
TOP	(STACK)-->	ITEM
EMPTY	(STACK)-->	BOOLEAN

Axioms:
(stk is type of stack, itm is of type ITEM)

(1) EMPTY(NEW) = true
(2) EMPTY (PUSH(stk,itm))= false
(3) POP (NEW)= error
(4) TOP(NEW)= error
(5) POP(PUSH (stk.itm))= itm
(6) Top(PUSH(stk,itm)== BOOLEAN.

Algebraic specification of the LIFO property;
On the above algebraic specification:

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 result in an error.
4. There is no top item on a new stack.
5. Pushing an item onto stack and immediately popping if off leaves the stack unchanged
6. Pushing an items onto a stack and immediately requesting the top item returns the items just pushed onto the stack.

(iv)Regular Expression:

- ✓ Regular expression can be used to specify the syntactic structure of symbol strings.
- ✓ Because many software products involve processing of symbol strings, regular expression provide powerful and widely used notations in software engineering.
- ✓ Every set of symbols string specified by a regular expression defines a formal language.
- ✓ Regular expression can thus be viewed as language generators.

The rules for forming regular expression are quite simple:

(1) Atoms: The basic symbols in the alphabet of interest from regular expression.
(2) Alternation: If R1 and R2 are regular expression, then (R1|R2) is a regular expression.
(3) Composition : If R1 and R2 are regular expression then (R1 R2) is a regular expression ,
(4) Closure: If R1 is a regular expression, then (R1)* is a regular expression.

(5) Completeness: Nothing else is a regular expression.

An alphabet of basic symbols provides the atoms. The alphabet is made up of whatever symbols are of interest in the particular applications. Alternation, (R1|R2),denotes the union of the languages (sets of symbols strings) specified by R1and R2 and composition, (R1 R2) denotes the language formed by concatenation string from R2 onto string from R1 . Closure , (R1)*,denotes the language formed by concatenating zero or more string from R1 with zero or more string from R1.

Observe that rules 2,3 and 4 are recursive ie., they define regular expression in terms of regular expression. Rule 1 is the basis rules, and rule 5 completes the definition of regular expression.

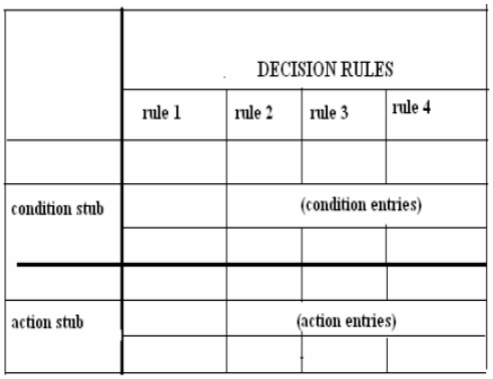
1.4.2 State – Oriented Notations:

State Oriented Notations consist of :-

(i) Decision Tables
(ii) Event Tables
(iii)Transition Tables.
(iv) Finite State Mechanism
(v)Petri nets.

(i)Decision Tables :

- Decision Tables provides a mechanism for recording complex decision logic.
- Decision tables are widely used in data processing applications and have an extensively developed literature.



Basic elements of a decision table

- The decision table is segmented into four quadrants:
  (a)condition stub -The Condition Stub contains all of the conditions being examined.
  (b)condition entry -The Condition entries are used to combine conditions into decision rules .
  (c) action stub - The Action Stub describes the actions to be taken in response to decision rules
  (d)action entry -The Action Entry quadrants relates decision rules to actions .

(ii) Event Tables:

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

A Two – Dimensional Event Table

- Its specify actions to be taken when event occur under different sets of conditions.
- A two dimensional event table relates actions to two variables.

  F (M, E) =A
  - ✓ M denotes the current set of operating conditions.
  - ✓ E is the event of interest
  - ✓ A is the actions to be taken.

If a system is in startup mode (SU) and event E13 occur action A16 is to be taken f(SU, E13) =A16.

Actions separated by semicolon denote (A14; A32) might denote A14 followed sequentially by A32

(iii) Transition Table:

- It is used to specify changes in the state of a system.
- The State of the System summarizes the status of all entities in the system at a particular time.
- Given the current state and the current conditions, the next state results.:if in sate Si, condition Cj results in a transition to state Sk, we say f(Si,Cj) = Sk

	current input	
current state	*a*	*b*
s0 *s1*	*s0* *s1*	*s1* *s0*
	next state	

A simple transition table

- Transition diagrams are alternative representation for transition tables
- In Transition diagram,state becomes nodes in a directed graph and transition are represented as arcs between nodes
- Arcs are labeled with conditions that cause transitions.
- From the Two dimensional table only we can illustrate the corresponding transition table.
- Transition diagrams and transition ables arerepresentations for Finite State Automata .
  The Theory of Finite State Automata is rich, complex, and highly developed.

Present State	Input	Action	Output	Next State
S0	a			S0

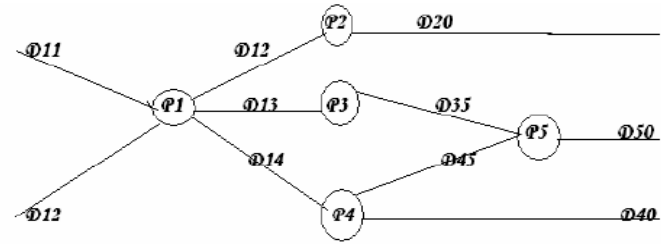
	b			S1
S1	a b			S0 S2

An augumented transition table

- ✓ Decision tables,Event tables, and transition tables are notation for specifying action as afunctions ofthe conditions that initiate those actions .
- ✓ Decision tables specify actions in terms of complex decision logic,event table relates actions to system conditions and transition tables incorporate the concept of system state.
- ✓ The notation are of equivalent expressive power;a specification in one ofthe notation can be readily expressed in the other two .
- ✓ The best choice among the tabular for requirements specification depends onthe particular being specified.

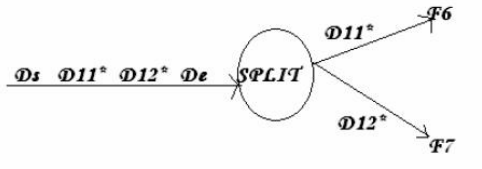
(iv) Finite State Mechanism:

- Data flow diagrams, regular expressions, transition tables are combined in finite state mechanism. For functional specification of software system.



A network data stream and process

- The data flow diagram for a software system consisting of a set of process interconnected by data streams.
- Data streams are specified using regular expressions.
- Process can be described using transition table.



Specification of the " split " process

Present State	Input	Actions	Output	Next State
S0	Ds	Open F6 Open F7		S1
S1	D_{11}	Write F6	D_{11}:F6	S1
	D_{12}	Close F6	D_{12}:F7	S2

		Write F7		
	D_E	Close F6 Close F7		S0
S2	D_{12}	Write F7	D_{12}:f7	S2
	D_E	Close F7		S0

- Processes split states in initial state so and wait for input D3.
- In state S2 split writes zero or more D12 messages to F7, then on receipt of the end of data marker D_E. Closes F7 and returns to state so to wait for next transmit ion.

Limitations

- Finite state mechanism is not possible for complex system.
- Because it involves large no of states and many combination of input data.

(v) Petri Nets:

- Petri nets were invented in the 1960s by Carl Petri.They provide a graphical representation technique and systematic and systematic methods.
- It was invented to overcome the limitations finite state mechanism.
- Concurrent systems are designed to permit simultaneous execution of the software.
- Components called task or process on multiple processors.
- Concurrent task must be synchronized to permit communication among task that operates at different execution rates, to prevent simultaneous updating of shared data and to prevent deadlock.
- Deadlock occurs when all the task in a system are waiting for data or other resources that can also waiting on other task.
- Fundamental problems of concurrency are synchronization, mutual exclusion and deadlocks.
- A pertinent is represented as a misdirected graph
- Two types of nodes are used in a petrinets called (1) places and (2)transition.

Places:

- ✓ Places are marked by tokens.
- ✓ Petrinets are characterized by an initial marking of places and a firing rule.
- ✓ A firing rule has two aspects.

Transition:

- ✓ A transition is enabling if every input place has at least one token.
- ✓ An enable a transition can fire.
- ✓ When a transition fires each input place of that transition looses one token and each output place of that transition gains one token.
- ✓ A marked petrinets is commonly defined as a quadruple -4 values in a system.
- ✓ Consisting of a set of places P, as set of transition T, a set of arcs A and a marking
- ✓ M .C= (P, T, A, M).where

P ={p1,p2,…….pm}

T= {t1,t2,……..tn}

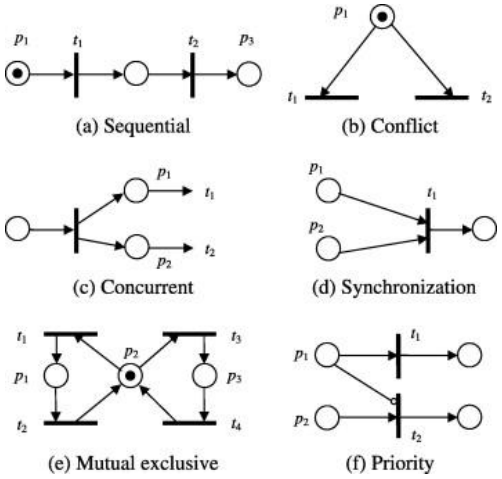
In the figure given below, Petri net models the indicated computation; each transition in the net corresponds to a task activation and places are used to synchronize processing.

Completion of t0removes the initial token from p0 and places a token on p1, which enables the co-begin.

Firing of the co-begin removes the token from p1 and places a token on each of p2, p3, and p4. This enables t1, t2, and t3; they can fire simultaneously or in any order.

This corresponds concurrent execution of taskst1, t2, t3.When t1, t2, t3 completes its processing, a token is placed on the corresponding output place p5, p6 or p7.

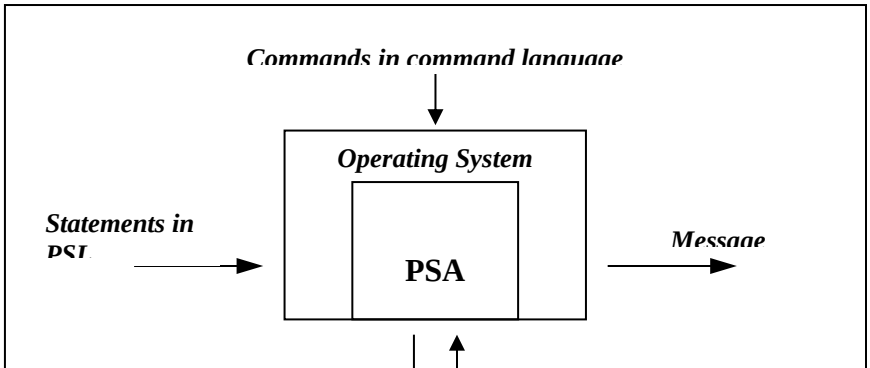
Co-end is not enabled until all the three tasks complete their processing. Firing the co-end removes the tokens from p5, p6, and p7 and places a token on p8, thus enabling p4, which can fire.



A deadlock situation occurs, when both t1 and t2 are waiting for the other to fire and neither can proceed.

Conflict Petri nets provide the basis for modeling of mutual exclusion. Both t1 and t2 are enabled, but only one can fire. Firing one will disable the other.

LANGUAGES AND PROCESSORS FOR REQUIREMENT SPECIFICATION



Structure of the Problem Statement Analyzer

PSL /PSA

The Problem Statement Language (PSL) was developed by Professor Daniel Teichrow. PSA – the Problem Statement Analyzer is a PSL processor.PSL and PSA were developed as components of the ISDOS project.

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 interconnected; the connections are called relationships. The general model is specialized to information systems by allowing only limited number of predefined objects, properties and relationships.

In PSL, system description can be divided into eight aspects:

- System input/output flow -The system input/output flow aspects deals with the interaction between the system and its environment.
- System structure - concerned with the hierarchies among objects in a system
- Data structure - includes all the relationships that exist among data used and manipulated.
- Data derivation - specifies which data objects are involved in particular processes
- System size and volume – concerned with the size of the system and the factors which affect size
- System dynamics – presents the manner in which the system "behaves"
- System properties – presents the system properties
- Project management - involves identification of the people, their responsibilities, schedules and cost estimates.

PSA operates on a database of information collected from a PSL description. The PSA system can provide reports in four categories:

- Database modification reports – database modification with diagnostic and warning signals
- Reference reports - includes Name List Report , The Formatted Problem Report, The Dictionary Report and data dictionary

- Summary reports - includes Database Summary Report, The Structure Report and External Picture Report
- Analysis reports – includes Contents Comparison Report, Data Processing Interaction Report and The Processing Chain Report.

PSL/PSA is a useful tool for documenting and communicating software requirements. PSL/PSA not only supports requirements analysis; it also supports design. This is mainly criticized for not incorporated any specific problem solving technique. PSL/PSA has been used in many different situations, ranging from commercial data processing applications to air defense systems.

RSL/REVS

The Requirements Statement Language (RSL) was developed to permit requirements for real-time software systems.

The Requirements Engineering Validation System processes and analyzes RSL statements. The fundamental characteristic of RSL is the flow-oriented approach used to describe real-time systems.

Flows are specifies in RSL as requirements networks(R-NETS). R-NETS have both graphical and textual representation.

- The Requirements Statement Language (RSL) was developed by the TRW defense and space systems group to permit concise unambiguous specification of requirements for real-time software for real-time software systems.
- The fundamental characteristics of RSL are the flow-oriented approach 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.
- The flow approach also provides for direct testability of requirements. A system can be tested to determine the responses are a specified under various stimuli, and performance characteristics and validation conditions can be associates with particular points in the processing sequence.

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

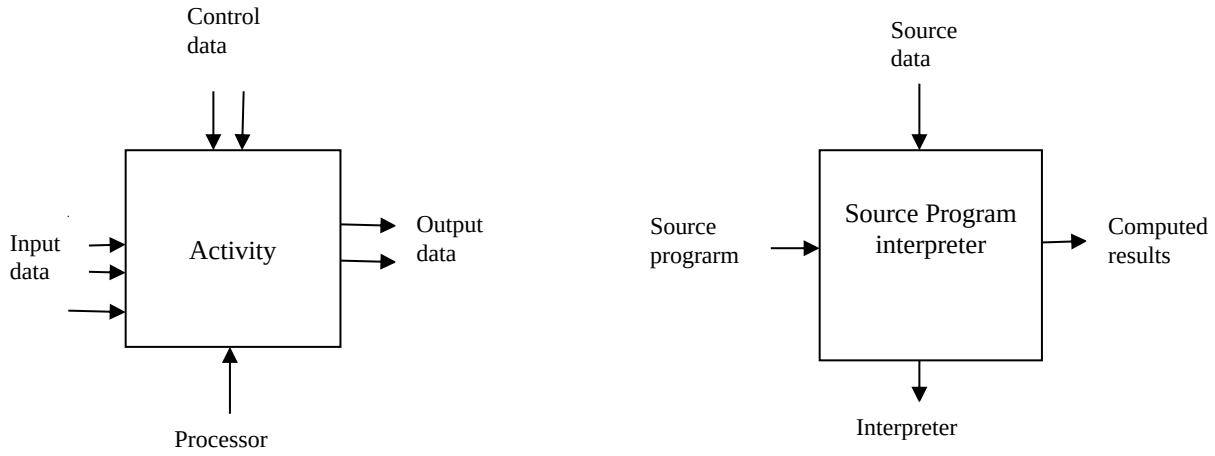
1. A translator for RSL
2. A Centralized database, the Abstract System Semantic Model(ASSM)
3. A set of automated tools for processing information in ASSM

Structured Analysis and Design Technique (SADT)

SADT was developed by D.T. Ross which incorporates a graphical language and a set of methods and management guidelines for using the language.

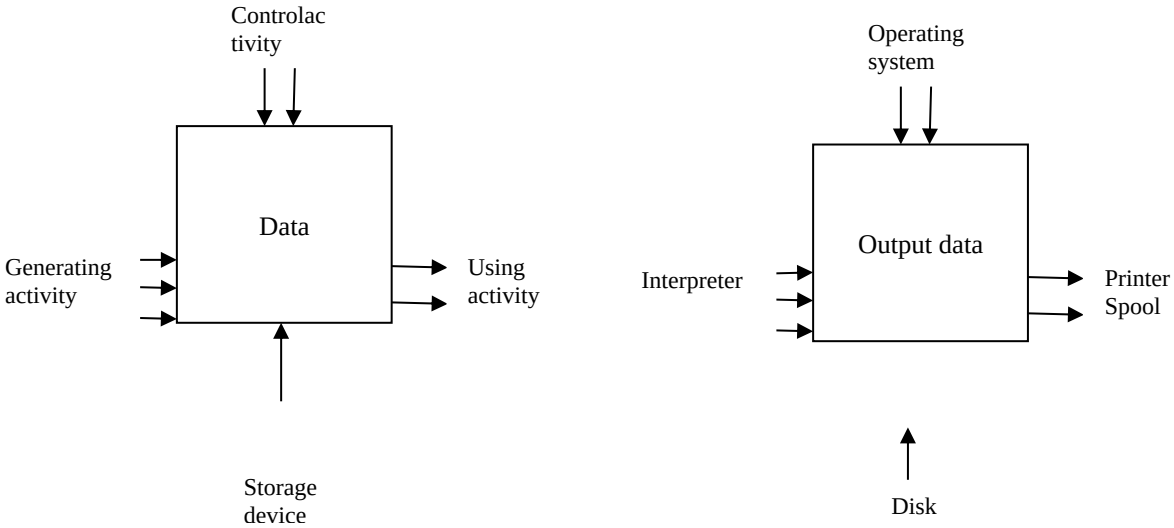
An SADT model consists of an ordered set of SA diagrams. Two basic types of SA diagrams are the activity diagram (actigram) and data diagram (datagram).

On an actigram the nodes denote activities and the arcs specify data flow between activities. Datagrams specify data objects in the nodes and activities on the arcs.



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.



Data diagram Components

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 devices used to store the representations of data objects. Control in datagram controls the conditions under which the node will be activated.

- Each node in an SA diagram can be expanded to indicate hierarchical relationships.
- External inputs, outputs, controls and mechanisms of a node which is a decomposition node are restricted to all these arcs of the original node.
- All the arcs incident on the node in the parent diagram must be depicted in the expanded diagram.

- SADT is versatile and can be applied to a wide variety of problems.
- SADT provides a notation and a set of techniques for understanding and recording complex requirements in a clear and concise manner.

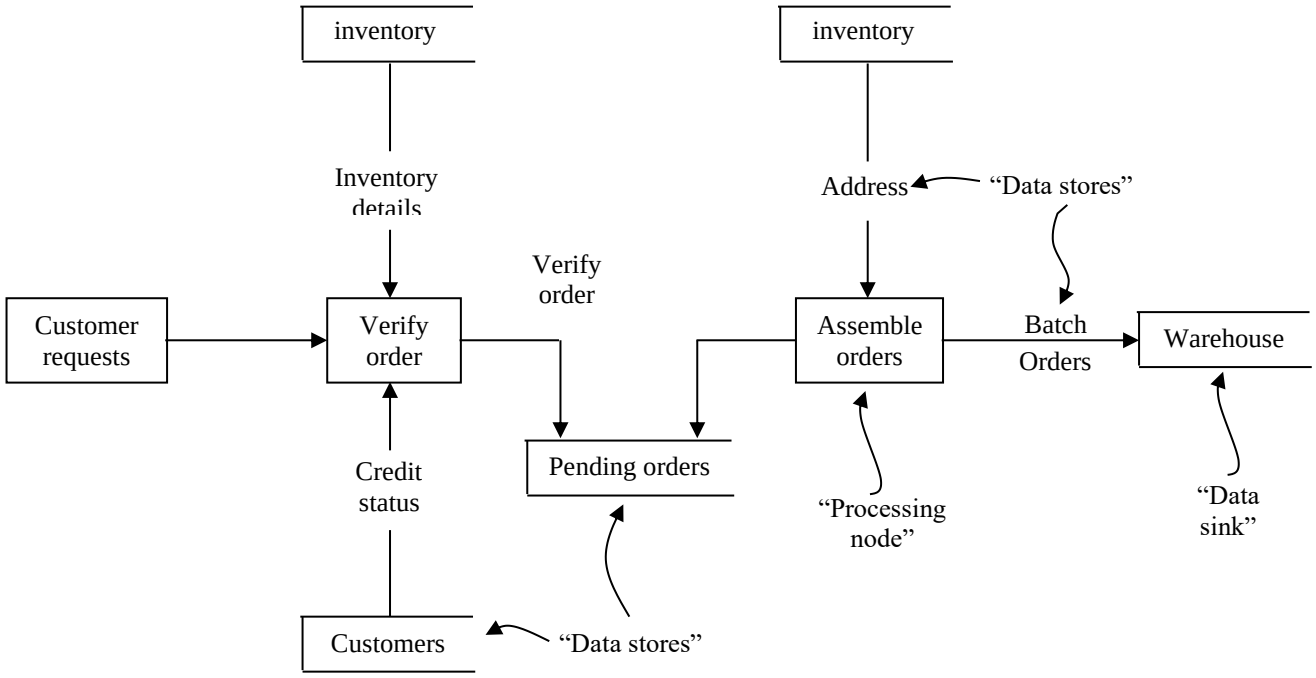
Structured System Analysis (SSA)

Structured System Analysis developed by Gane and Sarson is database concepts whereas SSA model developed by DeMarco does not include database concepts. Gane and Sarson's SSA has four basic features.

- Data flow diagrams
- Data dictionaries
- Procedure logic representations
- Data store structuring techniques

In a data flow diagram open-ended rectangles indicate data stores, labels on the arcs denote data items, shaded rectangles depict sources and sinks for data and the remaining rectangles indicate processing steps.

Order processing:



A data flow diagram (adapted from (GAN79))

- A data flow diagram do not indicate mechanism and control and an additional notation is used to show data stores.
- A data dictionary is used to define and record data elements.

- Processing logic representations such as decision tables are used to specify processing sequences in terms that are understandable to customers and developers.
- SSA model uses a relational model to specify data flows and data stores. Relations are composed from the fields of data records.

ORDER("CUSTOMER_IDENTITY, ORDER_DATE, CATALOG_NUMBER,ITEM_NAME," UNIT_PRICE, QUANTITY, TOTAL_COST)

.

Gist

Gist is a formal specification language developed by R.Balzar. Gist is a textual language based on relational model of objects and attributes.

A Gist specification is composed of three parts.

1. A specification of object types and relationships. This determines possible states.
2. A specification of actions which define transition between possible states.
3. A specification of constraints on states and state transitions.

Preparing Gist involves the following steps.

1. Identify a collection of object types manipulated by the process.
2. Identify individual objects (values) within types.
3. Identify relationships to specify the ways in which objects can be related to one another. Typical relationships include "connects to ", "is part of", "derived from"
4. Identify types and relationships that can be defined by recursive definition in terms of other types and relations.
5. Identify static constraints on the types and relationships.
6. Identify actions that can change the state of the process in some way.
7. Identify dynamic constraints.
8. Identify active participants in the process being specified and group them into classes, such that common actions are performed by the participants in a class.

The Initial Operating Capability (IOC) is a prototype testing facility for software specification written in Gist. Gist is a well defined but complex syntax.

SOFTWARE ENGINEERING
UNIT III

SOFTWARE DESIGN

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, architectural design, and detailed design.

- **External design** of software involves conceiving, planning out, and specifying the externally observable characteristics of a software project.
- These characteristics include user displays and report formats, external data sources and data sinks, and the functional characteristics performance requirement and high-level process structure for the project.
- **Internal design** involves conceiving planning out, and specifying the internal structure and processing details of the project.
- **Architectural design** is concerned with refining the conceptual view of the system, identifying internal processing functions, decomposing high level functions into sub functions, defining internal data streams and data stores, and establishing relationships and interconnections among functions, data streams, and data stores.
- **Detailed design** includes specification of algorithms that implement the functions.

4.1.Fundamental Design Concepts

- Fundamental principles provide the underlying basis for development and evaluation of techniques fundamental concepts of software design include
    - ***i. abstraction,***
    - ***ii. structure,***
    - ***iii. information hiding,***
    - ***iv. modularity,***
    - ***v. concurrency,***
    - ***vi. verification, and***
    - ***vii. design aesthetics***

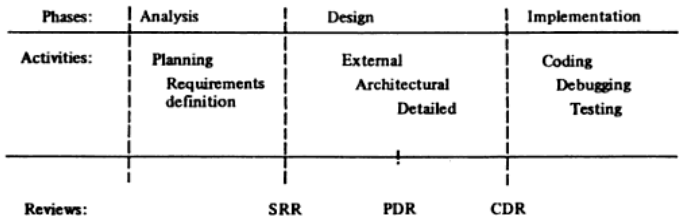


Figure 5.1 Timing of the Software Requirements Review (SRR), the Preliminary Design Review (PDR), and the Critical Design Review (CDR).

(i) Abstraction

- Abstraction is the intellectual tool that allows us deal with concepts apart from particular instances of those concepts.

- During requirements definition and design abstraction permits separation of the conceptual aspects of a system from the (yet to be specified) implementation details.
- We can, for example, specify the FIFO property of a queue or the LIFO property of a stack without concern for the representation scheme to be used in implementing the stack of queue.
- Three widely used abstraction mechanisms in software design are functional abstraction, data abstraction, and control abstraction.
- **Functional abstraction**allows the visible routines communicate with other groups and the hidden routines exist to support the visible ones.
- **Data abstraction** involves specifying a data type or a data by specifying legal operations on objects; representation and manipulation details are suppressed.
- Thus, the type "stack" can be specified abstractly as a LIFO mechanism.
- **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 abstraction of machine code implementations that involves conditional jump instructions.

(ii) Information Hiding

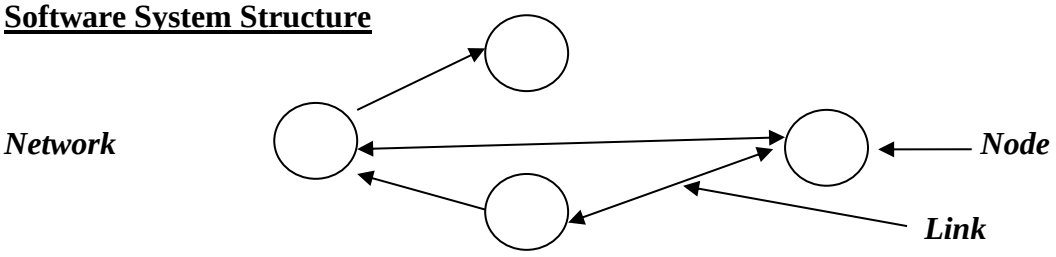
- Information hiding is a fundamental design concept for software. The principle of information hiding was formulated by Parnas. When a software-system is designed using the information hiding approach, each module in the system hides the internal details of its processing activities and modules communicate 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 it (this is the principle of data abstraction)
    2. the format of control blocks such as those for queues in an operating system (a "control-black" module)
    3. character code, ordering of character sets, and other implementation details.
    4. shifting, masking, and other machine dependent details.

(iii) Structure

- Structure is a fundamental characteristic of computer software. The use of structuring permits decomposition of a large system into smaller, more manageable units with well-defined relationships to the other units in the system.
- The most general from of system structure is the network. A computing network can be represented as a directed graph, consisting of nodes and arcs.
- The nodes con represent processing elements that transform data and the arcs used to represent data links between nodes.
- Hierarchical structure may or may not from tree structures.
- A Hierarchical structure isolates software components and promotes ease of understanding, implementation, debugging, testing, integration, and modification of a system.
- In hierarchical system

- There are usually one or more group of routines, designated as utility groups, these group typically contain lowest level routines, (routines that do not use other routines) that are used throughout the hierarchy.

Software System Structure



(iv) Modularity

A module is a FORTRONsubroutine or a module is an Ada package or a module is a work assignment for an individual programmer.

1. Each processing abstraction is a well-defined subsystem that is potentiallyuseful in other application.
2. Each function in each abstraction has a single, well-defined purpose.
3. Each function manipulates no more than one major data structure.
4. Function share global data selectively. It is easy to identify a routine thatshare a major data structure.
5. Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing,documenting, and maintenance of the software product.

(v) Concurrency

- Software system 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 have independent processes that can be activated simultaneously.
- Problems unique to concurrent systems include deadlock, mutual exclusion, and synchronization of processes.
- Deadlock is an undesirable situation that occurs when all processes in a computing system are waiting for other processes to complete some actions so that each can proceed.
- Mutual exclusion is necessary to ensure that multiple processes do not attempt to update the same components of the shared processing state at the same synchronization is required.

As hardware becomes more sophisticated, it becomes necessary of every software engineer to understand the issues and techniques of concurrent software design.

vi) Verification

- Verification is a fundamental concept in software design.
- A design is verifiable if it can be demonstrated that the design will result in an implementation that satisfies the customer'srequirements.
- This is typically done in two steps:

1. verification that the software requirementdefinitionsatisfies the customer's needs
2. verification thatthe design satisfies the requirements definition.

In software design, one must ensure that the test system is structured so that the internal states can be observed, tested, and the results related to the requirements

(vii) Aesthetics

- Aesthetics considerations are fundamental to design, whether in art or technology.
- Simplicity, elegance, and clarity of purpose distinguish of outstanding quality from mediocre products.

4.2 MODULES AND MODULARIZATION CRITERIA

Architectural design has the goal of producing well-structured, modular software system. A software module to be a named entity must have the following characteristics:

1. Modules contain instruction, processing logic, and data structures.
2. Modules can be separately 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 parameters.
5. Modules can use other modules

4.2.1 Coupling and cohesion

- There are numerous criteria that can be used to guide the modularization of asystem.
- A fundamental goal of software design is to structure the software product so that the number and complexity of interconnection between Modules is minimized.
- The strength of coupling between two modules is influenced by the complexity of theinterface.
- The type of connection and the type of communication.
- Modification of a common data block or control block may require modification of allroutines that are coupled to that block.
- Connections established by referring to other module names are more loosely coupled than connections established by referring to the internal element of 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

1. Content coupling:

Content coupling occur when one module modifies local data value or structure inanother module.

2. Common coupling

In common coupling, modules are bound together by global data structure.

3. Control coupling

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

4. Stamp coupling

Stamp coupling is global data items are shared selectively among routines that require the data.

5. Data coupling

Data coupling involves the use of parameter lists to pass data items between routines.

Cohesion

The internal cohesion of a module is terms of the strength of binding of elementswithin the module. Cohesion of element occurs on the scale of weakest (least desirable) to strongest (most desirable) in the following order:

1. **Coincidental cohesion**
2. **Logical cohesion**
3. **Temporal cohesion**
4. **Communication cohesion**
5. **Sequential cohesion**
6. **Functional cohesion**
7. **Informational cohesion**

1. **Coincidental cohesion**
   Coincidental cohesion occurs when the elements within a module have no apparent relationship to one another.
2. **Logical cohesion**
   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 audits all data.
3. **Temporal cohesion**
   Modules with temporal cohesion exhibit many of the same disadvantages as logically bound modules. A typical example of temple of temporal cohesion is a module that performs program initialization. In higher on the binding scale than temporal binding because the elements are executed at one time and also refer to the same data.

4. Communication cohesion

The elements of a module possessing communicational cohesion refer to the same set of input and or output data. For Eg:- "Print and punch the output File" is communicationally bound. Communicational binding is higher on the binding scale than temporal binding because the elements are executed at once time also refer to the same data

5. Sequential cohesion

Sequential cohesion of elements occurs when the output of one element is the input for the next element.

6. Functional cohesion

Functional cohesion is a strong, and he desirable, type of binding of elements in a

module because all elements are related to the performance of a single function.

7. Informational cohesion

This occurs in module with complex processing steps and data structures.Informational-bound modules contain routines that are functionally bound.This is similar to communicational cohesion.

Here only the functionally cohesive routine of the whole module is executed when the module is called.

The cohesion of a module can be determined by writing a brief statement of purposefor the module and examining the statement .The following tests are suggested byConstantine:

1. If the sentence has to be a compound sentence containing a comma or containing more than one verb, the module is probably performing more than one function; therefore, it probably has sequential or communicational binding.
2. If the sentence contains words relate ting to time, such as "first," "next," "than," "after," "when," "start," than the module probably has sequential or temporal binding.
3. If the predicate of the sentence does not contain a single, specific object following the verb, the module is probably logically bound. For example "edit all data" has logical binding; "edit source data" may have functional binding.
4. Words such as "initialize" and "clean up" imply temporal binding. In summary, the goal of modularizing a software system using in the coupling cohesion criteria is to produce systems that have stamp and data coupling between the module, and functional or informational cohesion of elements within each module.

4.3 DESIGN NOTATIONS

In software design, as in mathematics, the representation schemes use are offundamental importance. Good notation can clarify the interrelationships andinteractions of interest, while poor notation can complicate and interfere with gooddesign practice.

Notations used to specify the external characteristics, architectural structure and processing details of a software system include data flow diagrams, structure charts, HIPO diagrams, procedure specification, pseudo code, structured English, and structured flowcharts.

4.3.1 Data Flow Diagram :-

- Data flow diagrams ("bubble charts") are directed graphs in which the nodes specify processing activities and the arcs specify data items transmitted between processing nodes.
- A data flow diagram might represent data flow between Individual statement or blocks of statement in a routine, data flow between sequential routine, data flow between

concurrent processes, or data flow in a distributed computing system where each node represents a geographically remote processing unit.

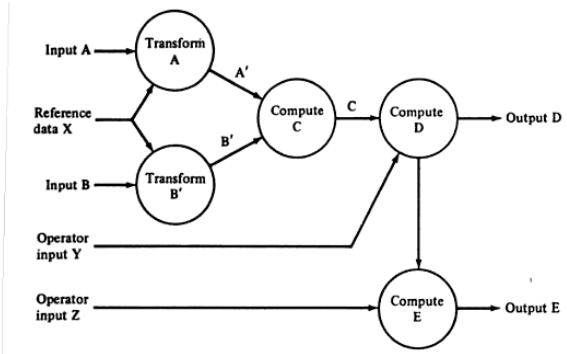


Figure 5.6a An informal data flow diagram or "bubble chart."

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

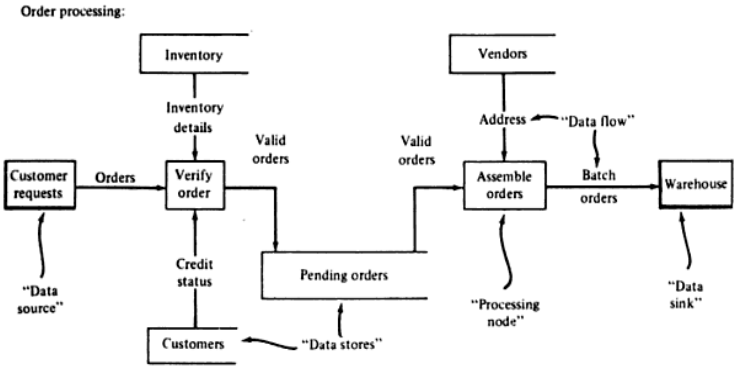
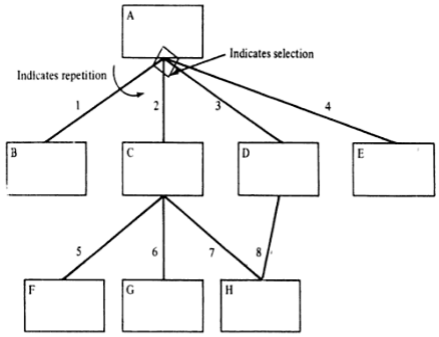


Figure 5.6b. A formal data flow diagram *(adapted from GAN79)*.

4.3.2 *Structure Charts*

- Structure charts are used during architectural design to document hierarchical structure, parameters, and interconnections in a system.
- A structure charts differ from a flowcharts in two ways: a structure chart has no decision boxes, and the sequential ordering of tasks inherent in a flowchart can be suppressed in a structure chart.



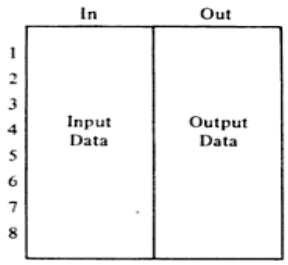


Figure 5.7 Format of a structure chart.

4.3.3 HIPO Diagrams:-

HIPO diagrams (hierarchy-process-input-output) were developed at IBM as design representation schemes foe top-down software development, and as external documentation aids for released products.

- A set of HIPO diagrams contains a visual table of contents, a set of overview diagrams, and of detail diagrams.
- The visual table content is a directory to the set of diagrams in the package; it consists of a tree-structured 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 charts.
- Overview diagrams specify the functional processes in a system.
- Each overview diagram describes the inputs, processing steps, and outputs for the function being specified.
- An overview diagram can point to several subordinate detail diagrams, asrequired.
- Detail diagrams have the same format as overview diagrams.

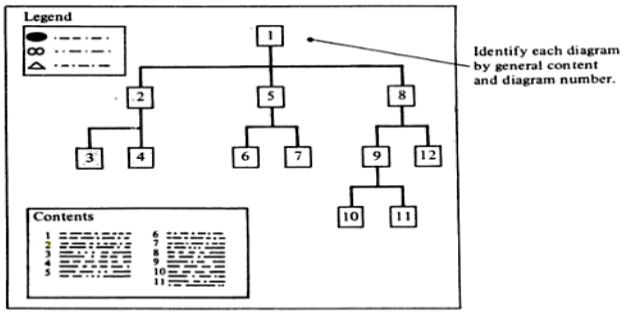


Figure 5.8a Visual table of contents for a HIPO package.

4.3.4 Procedure Templates:-

- In the early stages of architectural 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.
- Mortifications to global variables, reading or writing a file, operating or closing a file, or calling a procedure that in turn exhibits side effects are all examples of side effects.

- During detailed design, the processing algorithms and data structures can be specified using structured flowcharts, pseudo code, or structured English.
- The syntax utilized to specify procedure interface during detailed design may, in fact, be the syntax of the implementation language.
- In this case, the procedure interface and pseudocode procedure body can be expanded directly into source code during product implementation

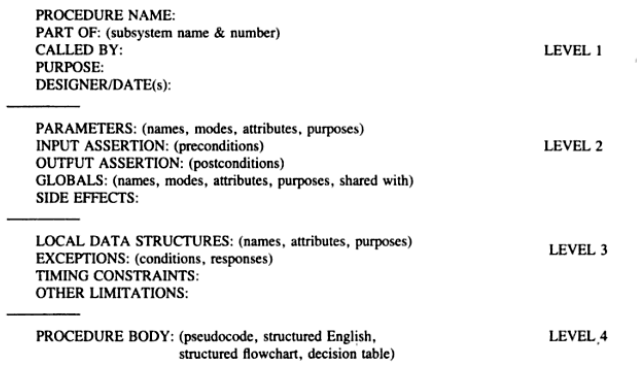


Figure 5.10 Format of a procedure template.

4.3.5 Pseudo Code

- Pseudo code notation can be used in both the architectural and detailed design phases. Like flowcharts, pseudo code can be used at any desired level of abstraction. Using pseudo code, the designer describes system characteristics using short.
- Concise, English phrases that are structured by key words such as if-than-else, while-do, and end. Key words 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

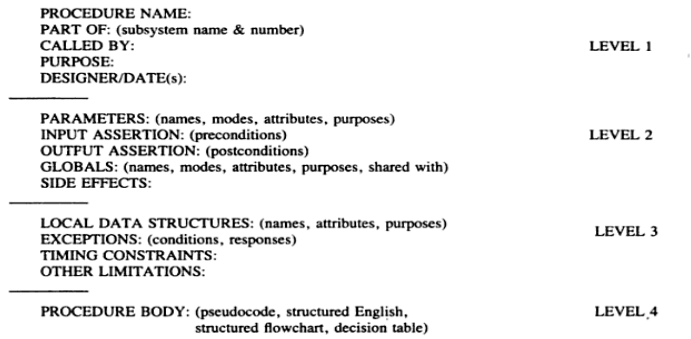


Figure 5.10 Format of a procedure template.

4.3.6. Structured Flowcharts:-

- Flowcharts are the traditional means for specifying and documenting algorithmic details in a software system.
- Flowcharts incorporate rectangular boxes or actions, diamond shaped boxes for decisions, directed arcs for specifying interconnections between boxes, and a variety of specially shaped symbols to denote input, output, data stores, etc.
- Structured flowcharts are logically equivalent to pseudo code, they have the same expressive power as Pseudo code; both can be used to express any conceivable algorithm.
- Structured flowcharts emphasize flow of mechanisms due to the graphical nature of the visual image.
- They are thus appropriate when decision mechanisms and sequencing of control flow are to be emphasized.

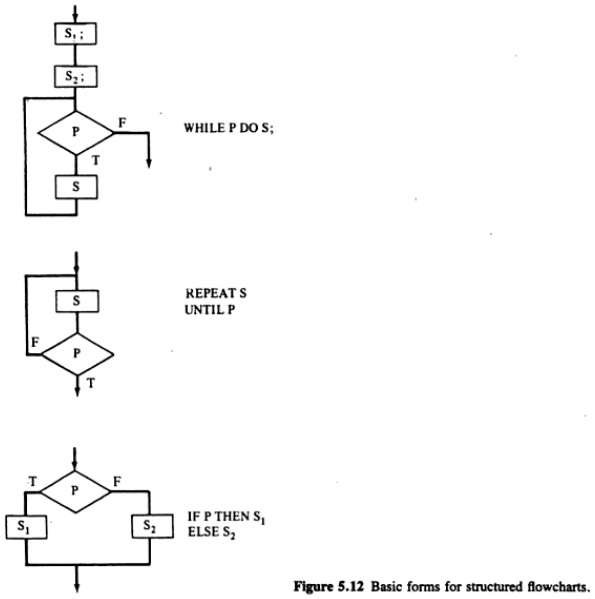


Figure 5.12 Basic forms for structured flowcharts.

4.3.7. Structured English

- Structured English can be used to provide a step-by-step specification for an algorithm like pseudo code, structured English can be used at any desired level of detail.
- Structured English is often used to specify cookbook recipes:
    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.

An example of using structured for software specification is provided. where it is used to specify the processing part of the HIPO diagram.

4.3.8 Decision tables

- Decision tables can be used to specify complex logic in a high-level softwarespecification, they are also useful for specifying algorithmic logic during detaileddesign.
- At this 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.

4.4 DESIGN TECHNIQUES

The design process involves developing a conceptual view of the system,establishing system structure, identifying data streams and data stores, decomposinghigh level function into sub function, establishing relationships and interconnectionsamong components, developing concrete data representation, and specifyingalgorithmic details.

Developing a conceptual view of a software system involves determining the type of system to be built.

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 appropriate application areas .

A data store is a conceptual data structure, .during external and architectural design data structures should be defined as data abstraction with emphasis placed on the desire operations and not on implementation details.

Data streams and data stores can be specified using data flow diagrams data dictionaries and data abstraction techniques.

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 .

Design techniques are typically based on the “top-down” and /are “bottom-up” design strategies .using the “top-down” approach, attention is first focus on global aspects of the overall system .as the design progress ,the system is decomposed into subsystem and more consideration is given to specific issues. backtracking is fundamental to “top-down” design.

In the “bottom-up” approach to software design, the designer first attempts to identify a set of primitive objects , action and relationships that will provide a basis for problem solution .higher-level.

Concepts are then formulated in term of the primitives. The “bottom-up” strategy requires the designer to combine features provided by the implementation language into more sophisticated entities.

Bottom-up design and implementation permits assessment of subsystem performance during system evolution. Top-down design and implementation permits early demonstration of functional capabilities at user level.

4.4.1 Stepwise refinement involves the following activities.

Stepwise refinement is a Top-down technique for decomposing a system from high-levelspecification into more elementary levels .

1. Decomposing designs 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 refine process is a faithful expansion of previous steps

Stepwise refinement 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 problem. The process is then repeated for each part of the system until it is decomposed in sufficient detail so that implementation in an executable programming language is straightforward.

The following example, adapted from Wirth(WIR73),illustrates detailed design by the method of successive refinement. A routine is required to write the first N prime numbers into file F.N is a formal parameter in the routine, and F is globally known

The initial version :

```
varI,X :integer;
begin rewrite(f);x:=1;
for I: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 Boolean variable, PRIM, the statement can be expressed as

```
Refinement1:  repeat X:=X+1;
              PRIM:=”X is a prime number”
              Until PRIM;
```

All prime numbers except the (which is 2)are odd. We can treat 2 as a special caseand increment X by 2 in the repeat loop.

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(formally or informally)

4.4.2 Levels of Abstraction :

- Levels of abstraction was originally described by Dijkstra as a layering of hierarchical Levels starting at level 0(processor allocation, real-time clock interrupts)and building up to the level of processing independent user programs.
- In Dijkstra’s system (The T.H.E. system).each level of abstraction is composed of a group of a group of related function, some of which are externally visible(can be invoked by function on higher levels of abstraction)and some of which are some of internal to the level.

- Internal functions are hidden from other levels; they can only be invoked by functions are used to perform tasks.
- Common to the work being performed on that level of abstraction.
- Of course, functions on higher levels cannot be used by functions on lower levels; function usage establishes the levels of abstraction.

Levels of Abstraction in the T.H.E.Operating System

- ✓ **Level 0 : Processor allocation clock interrupt handling**
- ✓ **Level 1 : Memory segment controller**
- ✓ **Level 2 : Console message interpreter**
- ✓ **Level 3: I/O buffering**
- ✓ **Level 4 : User Programs**
- ✓ **Level 5 : Operator**

4.4.3 Structured Design:

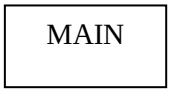
- Structured design was developed by Constantine as a top-down technique for architectural design of Software system.
- The basic approach in structured design is systematic conversion of data flow diagrams into structure charts.
- Design heuristics such as coupling and cohesion are used to guide the design process.

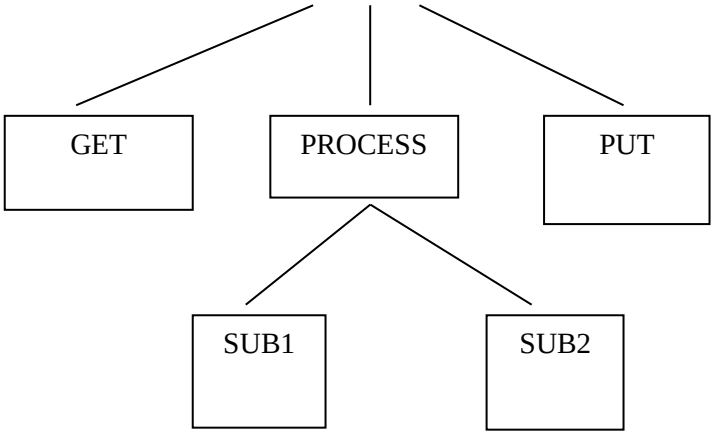
Steps of Structured Design :

- ✓ The first step in structured design is review and refinement of the data flow diagram(s) developed during requirements and external design .
- ✓ the second step is to determine whether the system is transform centered or transaction driven and to drive a high level structure chart based on this determination.
- ✓ In a transform centered system the data flow diagram contains input processing and output segment that are converted into input processing, and output subsystem in the structure chart.
- ✓ The third step in structure design is decomposition of each subsystem using guidelines such as coupling, cohesion, information hiding ,levels of abstraction ,data abstraction and other decomposition criteria .

4.4.4 Integrated Top-Down Development:

Integrated Top-down development strategy





- Integrated top-down development integrates design implementation and testing.
- Using integrated top-down development, design proceeds top-down from the highest-levelroutines; they have the primary function of coordinating and sequencing the lowerlevel routines.
- Lower-level routines may be implementation of elementary functions (Those that call no other routines), or they may in turn invoke more primitive routines.
- There is thus a hierarchical structure to a top-down system in which routines can invoke lower-level routines but cannot invoke routine on a higher level.
- The integrated top-down design technique provides an orderly and systematic frame work for software development.
- A further advantage of the integrated top-down approach is distribution of system integration across the project ;
- The interfaces are established, coded, and tested as the design progresses.

The primary disadvantage of the integrated top-down approach is that early high-leveldesign decisions may have to be reconsidered when the design progresses to lower levels .this may require design backtracking and considerable rewriting of code .

4.4.5 Jackson Structured Programming

Jackson structured programming was developed by Michael Jackson as systematic technique for mapping the structure of a problem into program structure to solve the problem (JAC75). The mapping is accomplished in three steps:

1. The problem is modeled by specify the input and output data structures using tree structured diagrams .
2. The input-output model is converted into a structural model for the program by identifying points of correspondence between nodes in the input and output trees .
3. The model of the program is expanded in detailed design model that contains the operations needed to solve the problem.

Input and output structures are specified using a graphical notation to specify data hierarchy, sequences of data, repetition of data items, and alternate data items.

The second step of the Jackson method involves converting the input and output structures into a structural model of the program.

The third step expands the structural model of the program into a detailed design model containing the operations needed to solve the problem.

Difficulties encountered in applying the Jackson method include structure clashes and the need for look ahead.

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 TO NET_MOVMENT
9. SUBTRACT MOVMNT FROM NET_MOVMNT

Look-ahead problems can be resolved by a technique called backtracking.Transformations are performed during detailed design and implementation to produce a system that can be implemented on conventional machine architectures

4.4.7 Detailed Design Considerations:

- Detailed design is concerned with specifying algorithmic 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 semantic issues and less concern with syntactic details than is implementation.
- Detailed design separates the activity of low-level design from implementation, just as the analysis and design activities isolate consideration of what is desired from the structure that will achieve the desire result.
- An adequate detailed design specification minimizes the number of surprises during product implementation.
- The detailed design representation may utilize key words from the implementation language to specify data representation.

4.5 Real Time And Distributed System Design

- According to Franta, a distributed system consists of a collection of nearly autonomous processors that communicate to achieve a coherent computing system (FRA81).

- Each processor possess a private memory, and processors communicate through an inter connection network.
- By definition, real-time systems must specified amounts of computation within fixedtime intervals. Real-time systems typically sense and control external devices, respond to a external events ,and share processing time between multiple tasks .
- A real-time network for process control may consists of several mini computers and microcomputers connected to one or more large processors .each small processor may be connected to a cluster of real-time devices.
- Several notations have been developed to support analysis design of real-time and distributed systems.
- They include RSL, SDLP, NPN, communication ports ,and the notation developed by Kramer and Cunningham.
- In summary , the traditional considerations of hierarchy , information hiding ,and modularity concepts for design of real-time systems.

4.6 TEST PLANS

- The test plan is an important, but often over looked, product of software design.A test plan prescribes various kinds of activities that will be performed todemonstrate that the software product meets its requirements.
- The test plan specifies the objectives of testing, the test completion criteria, the system integration plan methods to use on particular models, and the particular cases to be used.
- There are four types of tests that a software product must satisfy: functional tests,performance tests, stress tests, and structural tests .
- Functional test cases specify typical operating conditions typical input value, andtypical expected results.
- Performance test should be designed to verify response time (under various loads), execution time, throughput, primary and secondary memory utilization and traffic rates, on data channels and communication links.
- Stress tests are design to over load a system in various ways, such as attempting to sign on more than the maximum number of terminals, processing more than the allowed number of identifiers of static levels, or disconnecting a communication link .
- Structural tests are concerned with examining the internal processing logic of software system.

4.7 Milestones Walkthroughs and Inspections

- Development of intermediate work products provides 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 software design are the preliminary design review(PDR) And the critical design review (CDR).
- The major goal of PDR is to demonstrate that the externally observable characteristics and architectural structure of product will satisfy the customer requirements.
- The CDR is held at the end of detailed design and prior to implementation .among other things, CDR provides a final management decision points to build or cancel the system.
- The CDR is in essence the repeat of the PDR ,but with the benefit of additional design effort.

Walkthroughsand Inspection

- A structured walkthrough is an in – depth technical review of some aspect of a software system.
- A walkthrough team consists of four to six people. The person whose material being reviewed is responsible for providing copies of the review material to members of the walkthrough team in advance of the walkthrough season and Team members are responsible for reviewing the material prior to the season.
- The focus of a walkthrough is on detection of errors are not on corrective actions.
- Design inspections are conducted by team of trained inspectors who work from check lists of items to examine .special forms are used to record problems encountered.
- In one experiment, 67 percent of error found in a software product during product development Was found using design code inspection prior to unit testing.

SOFTWARE ENGINEERING
UNIT IV

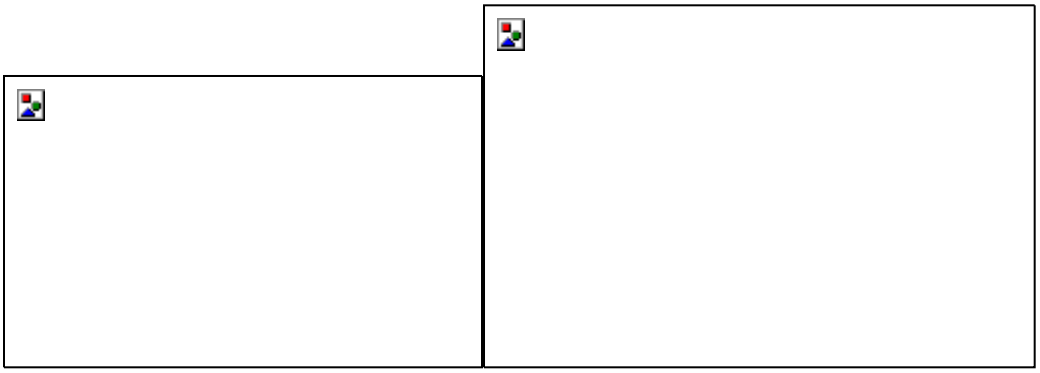
IMPLEMENTATION ISSUES

The use of structured coding techniques and styles supports the primary goal of implementation to write quality source code and internal documentation that aid in easy verification with specification, debugging testing and modification. The source code thus generated will be simple, clear and elegant and less obscure, clever and complex.

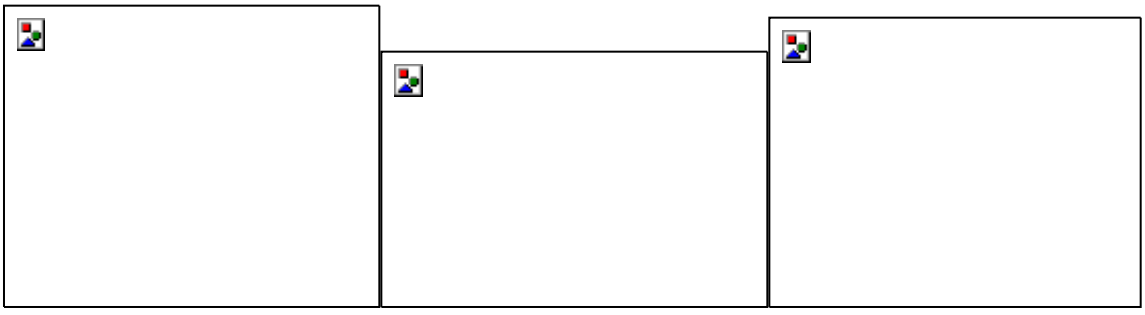
Structured coding techniques *aids in linearizing the control flow through a computer program so that the execution sequence follows the sequence in which the code is written.* This dynamic structure of a program as it executes them resembles the static structure of the written text which improves code-readability, debugging, testing, documentation and modification of programs. The following are some of the golden rules to be considered for structured coding.

Single entry, Single exit constructs

Single entry, single exit constructs are control structures that enables us to create sequencing, selection and iteration constructs. The term single entry, single exit is used since such constructs have a single entry or single path of inflow of data and a single exit or single path of outflow. The following are the common single entry, single exit constructs –



Sequencing Construct **Selection Construct**



Type 1: while-do looping ***Type 2:*** do-while looping ***Type 3:*** for loop

Iteration Construct

In cases when we require multiple loop exits, we may apply ‘go to’ as some languages allow it or should redesign the algorithm to avoid multiple loop exits, which is the desired practice. Alternatively in such cases, we may use the exit or ‘break’ statement as follows –

```
while (cond 1){
    Code 1;
    Code 2;
    if (Cond 2){
            break;
    }
}
```

Data encapsulation

Data encapsulation is the process of encapsulating the data operations of an object from outside interference or misuse. In object oriented languages such as C++, C# or Java, we implement data encapsulation by class abstractions. In such a case the class itself doesn’t have physical existence and is only a logical abstraction. When a class is instantiated, it creates objects with the encapsulated data and operations.

Goto statement

Goto statement provides unconditional transfer of control and thus allows violations of single entry,single exit conditions of strucrured coding.Thus it is a valuable mechanism when it is used in a disciplined and stylistic manner.

Recursion

A recursive subprogram is one that call itself either directly or indirectly.It is a powerful and elegant programming technique.The goals of clarity and efficiency are thus served by appropriate use if recursion.

Coding Style

The following are some of the major guidelines that dictates a perfect codig style

-DO –

3.1. Use a few standard control constructs – i.e, use limited number of standard control constructs such as those enabling sequencing, selection and iterations.

3.2. Use Gotos in a disciplined manner – Gotos may be applied in coding when multiple loop exits or control transfer to exception handling routines is un-avoidable. In all these cases, use of Goto statements should be in a stylish, disciplined manner to achieve the desired result. The acceptable use of Goto statements are almost always forward transfers of control within a local region of code and with traditional functional languages. Most of the modern programming languages provides various constructs such as exit, break, continue exception statements to transfer the control of execution without using goto.

3.3. Introduce user-defined datatypes to model entities in the problem domain – For eg:, C allows users to define data types of their own using the 'typedef' keyword as follows –
3.4. Hide data structures behind access functions. Isolate machine dependencies in a few routines
The purpose of isolating the machine dependencies in a few routines is to ease program portability.
3.5. Provide standard documentation prologues for each subprogram and/or compilation unit – A documentation prologue contains information about a sub-program or compilation unit that is not obvious from reading the source code.
3.6. Use indentation, parenthesis, blank spaces, blank lines and borders around blocks of comments to enhance code readability

– DONT –

3.7. Do not use programming tricks – Don't use programming tricks to too clever codes which may be inefficient and unstructured, un-readable and unmaintainable.
3.8. Avoid null-then statement – Null – Then statement is of the form
3.9. Avoid then – if statements
3.10. Avoid routines/methods that causes obscure side effects
3.11. Don't sub-optimize – Sub-optimization occurs when one devotes inordinate effort to refining a situation that has little effect on the overall outcome.
3.12. Carefully examine methods/routines having than 5 formal parameters – Methods/function with longer list of parameter values results in complex routines that are difficult to understand and difficult to use and they result from inadequate decomposition of a software system. Fewer parameters and fewer global variables improve the clarity and simplicity of code. In this regard, 5 is a very descent upper bound.
3.13. Don't use an identifier for multiple purposes – Programmers most frequently use a single identifier for multiple purposes for saving memory since as number of variables/identifiers is increased, the number of memory calls to be allocated is also increased. Such a practice is not recommended due to 3 reasons –
each identifier define a specific purpose and not various purposes
use of a single identifier for multiple purposes indicates multiple regions of code which leads to low cohesion
it makes the source code very sensitive to future modifications
3.14. Don't nest the loops too deeply – Deeply nesting loops will make it difficult to determine the conditions under which statement in the lowest loop will be executed and such a practice is a poor design. Nested loops exceeding 3-4 levels should be avoided.

Coding Guidelines

Coding standards are specifications for a preferred coding style.
Guidelines:

1) The use of goto statements should be avoided in normal circumstances.
2) The nesting depth of program should be less in normal circumstances.
3) The number of executable statements in a subprogram should be minimum.
4) Departure from normal circumstances need approval from project leader.

Documentation Guidelines

Computer software includes the source code for a system and all the supporting documents generated during analysis ,design,implementation,testing and maintenance of the system. Internal documentation includes std prologues for compilation units and subprograms ,the self documenting aspects of the source code and the internal comments embedded in source code.Some of the general guidelines for internal documentation is as follows –

Minimize the need for embedded comments by using

- Standard prologues
- Structured programming constructs
- Good Coding style

Descriptive names from the problem domain for user-defined data types, variables, formal parameters, enumeration literals, methods, files etc.

Self documenting features of the implementation language such as user-defined exceptions, user defined data types, data encapsulation etc.

Attach comments to blocks of code that

perform major data manipulations

simulate structured control constructs

perform exception handling

Use problem domain terminology in the comments

Use blank lines, borders, and indentation to highlight comments

Place comments to the far right to document changes and revisions

Type Checking •

Type checking is the activity of ensuring that the operands of an operator are of compatible types. • A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type. • This automatic conversion is called a coercion.

Five levels of type checking:

Level 0:Typeless

Level1:Automativc type coercion

Level2:Mixed mode

Level3:Psedocode strong type checking

Level4:Strong type checking

Level 0:Typeless

Some programming languages have no declaration statement.

Level1:Automativc type coercion

Attributes are implicitly converted, by compiler-generated code.

Level2:Mixed mode

Some programming languages permits mixed mode operations between similar data types.

Level3:Psedocode strong type checking,Level4:Strong type checking

In strong typed programming languages operations are permitted only between objects of equivalent types. Psedocode strong type checking is a strong type checking with loopholes.

SCOPING RULES:

The scoping rules dictate the manner in which identifiers can be defined and used by the programmer.

Scoping rules include global scope ,fortran,nested and restricted scope.

In global scope all identifiers are are known in all regions.

In Fortran scope the identifiers are known throughtout the containing program into not outside the program unit.

In nested ,programming units can be nested within other units.

CONCURRENCY MECHANISM

Two or more segments of a program can be executed concurrently if the effect of executing the egments is independent of the order in which they are executed.

The 3 fundamentals of concurrent programming :

Shared variables,Asynchronous message passing and Synchronous message passing.

UNIT V

VERIFICATION AND VALIDATION TECHNIQUES

Introduction:

The goals of verification and validation activities are to assess and improve the quality of the work products generated during development and modification of software.

Quality attributes of interest include correctness, completeness, consistency, reliability, usefulness, usability, efficiency, conformance to standards, and overall cost effectiveness.

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

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".

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.

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
5. Standards, practices, and conventions to be used
6. Reviews and audits to be conducted
7. A configuration management
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
13. Methods and facilities to be used in collecting, maintaining, and retaining quality assurance record.

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.

2. During product evolution, In-Process Audits are conducted to verify consistency and completeness of the work products.

3. Prior to product delivery, a Functional Audit and a Physical Audit are performed. The

Static analysis

Static analysis is a technique for assessing the structural characteristics of source code, design specifications, or any notational representation that conforms to well-defined syntactic rules.

Static analysis can be performed manually using walkthrough or inspection techniques;

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

Symbolic Execution

- This is a validation technique that assigns symbolic values to the input variables in a program rather than assigning literal values.
- The whole program from input to the output is represented in symbolic form.
- The operands in the expressions are assigned symbolic input values.
- The intermediate expressions as well as decisions are also expressed in symbolic form of input variables. Example :

Symbolic Execution

A ← a; B ← b;
C ← a +b;
D ← (a +b) * a;

IF[(a+b) * a] > b ….. condition

Sample Code Segment

```
Read (A, B);
C = A + B;
D = C* A;
IF (D>B) THEN
        -------
        -------
ELSE
        -------
```

- As seen above all the expressions and conditions are converted into symbolic form of inputs. Once the symbolic expressions are formed, then the path conditions for the IF statements in the

<u>Unit Testing and Debugging</u>

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.

The term "program unit" to denote a routine or a collection of routines implemented by an individual programmer.

Unit Testing

Unit testing comprises the set of tests performed by an individual programmer prior to integration of the unit into a larger system.

A program unit is usually small enough that the programmer who developed it can test it in great detail.

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

Functionaltests
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.

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.

Stress tests are those tests designed to intentionally break the unit.

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.

Debugging

Debugging is the process of isolating and correcting the causes of known errors. Commonly used debugging methods include induction, deduction, and backtracking.

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.
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.
4. Prove or disprove each hypothesis.
5. Implement the appropriate corrections.
6. Verify the correction

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.

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.

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests to verify that the implemented system satisfies its requirements.

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.

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.

Formal Verification

Certain mathematical methods are employed to verify whether the program under test satisfies certain conditions and possess the desired properties. The formal verification methods are :

- Input-Output Assertions
- Weakest Preconditions
- Structural Induction

1. Input – Output Assertions

- The contributors to this method were Floyd, Hoare and Djkstra.
- The basic concept in this method is to associate certain assertions or predicates or verification conditions to the entry point, exit point and other points of the source code and then verifying if these conditions are true at all times – when the program is executed.
- If P and R are predicates or conditions, S is a code segment then
- S ® implies P is true before the execution of S and R is true after the execution of S.
- Example : (1 <i<N) I = i + 1 (2 < i < N+1)
- Composition rule can be applied to the predicates and we get :

  IF (P) S1(Q) and (Q) S2(R)

  $\Rightarrow$ (P) S1; S1 (R)
- The above can be expressed in words as follows : if we intermediate conditions in an execution path of a program are true and the input conditions' true criterion implies the truth of the output predicate for that particular execution path.

- This also states that if the intermediate conditions from the input assertion to the output assertion are true and if the input conditions are satisfied and if the program runs and terminates through the execution path under consideration then the output assertion or condition will be satisfied (true) at the end of the program.
- In programs such as ones using loops the termination occurs when the variable used inside the loop reaches certain upper value or a lower value due to repeated cycles of addition or subtraction to or from its earlier value. This can be seen in programs used to find the sum of the first N natural numbers or to find the factorial of a number.
- Such a loop invariant must be true when the loop execution begins: it must be true at any point and at any number of traversals and it

2 Weakest Precondition

- This is another method to verify the predicates such the value of loop invariants.
- If (P) s ® is a proposition then we call P as the weakest precondition for S if it is the weakest condition that will make R true after the execution of S.
- This is represented as P = wp (S, R)
- To find P we use a reverse method and proceed from R.
- If S is a statement such as X = Y then we obtain the weakest precondition P by

  Wp(S, R) = wp (X = Y, R) = R (Y –>X)
- Example

  Wp (X = Y +2, X = 5) = (X =5 and Y + 2 -> X)

  = (Y + 2 = 5) or (Y=3)
- Hence we find Y =3 is the weakest precondition which makes X = 9 after the execution of the statement X = Y =2.

3 Structural Induction

- Structural induction is a formal verification technique based on the principle of mathematical induction
- To slow a proposition P as true
  - Prove P as true of the small or minimal values of the set S.
  - Assume P to be true or satisfied for all elements less than or equal to N element in S.
  - Using appropriate arithmetic operations prove P to be true for the N +1 element in S.
- Example

  P = Prove that sum of n natural numbers = n(n+1) /2

  When = 1,

  $$\sum_{j=1}^{n} j = 1(1+1)/2 = 1$$ which is true

  This step is the **BASIS**

  Assume P to be true for an element k P to be true. Here k is less then or equal to m+1.

$$\sum_{j=1}^{n} j = n(n+1)/2$$

This is the **INDUCTIVE HYPOTHESIS**

The final step is **INDUCTION**

$$\sum_{j=1}^{n} j = \sum_{j=1}^{n} Aj + (m+1)$$

= m(m+1)/2 + (m+1) ……by inductive hypothesis step.

= (m+1) (m+2)/2

- This principle of mathematical induction can be applied to recursive functions, data structures, tree traversal algorithms quick easily.

Software Maintenance

- Software maintenance is the phase that occurs after the delivery of the product to the customer.
- Typically, if the development phase spans 1 or 2 years then the maintenance phase spans 5 years.
- It is well established that software maintenance activities account for the larger portion of the total life cycle (70%).
- Hence we find software maintenance is a microcosm of software development cycle.
- The major activities performed during this phase are
    - Enhancements to existing features,
    - Correcting and fixing bugs in the product,
    - Customizing and adapting the product to the client's environment.
    - Updating user documents,
    - Enhancing performance of the system,
    - Improving user interfaces.
    - Configuring the system to different hardware configurations and various network protocols.
- The goal of this phase is "customer satisfaction." The end user's ease to use the software is the most important criteria to satisfy their needs. Hence the maintainer is involved with the toughest tasks of correcting and solving their problems with assistance from the change control board.
- Smaller problems may be corrected without much of difficulty in changing designs. However if the bugs are quite complex or the users' needs change then it may require analysis and redesigning of that module. So maintenance reinitiates development in such a case.
- The ease of maintaining a software product highly depends upon how well the product has been designed and developed. Also the product characteristics such as clarity, modularity, good documentation of source code and the supporting documents contribute to software maintainability.

- Once implementations is over and the system is accepted by the customer, a set of programmers who have played their part in the development stages and who are aware of the under-lying architecture and the business – logics are allocated the task of maintaining the product. Such programmers are called Maintenance Programmers.

Enhance Maintainability During Development

There are certain steps that have to be performed in order to enhance the maintainability or a software product. Maintenance requires systematic approach to tracking and analysis of change requests, redesign, implementation and re-documentation.

Analysis Activities
• Develop standards and guidelines
• Set milestones for the support documents
• Specify QA procedures
• Determine the future product enhancements
• Calculate the estimated resources needed for maintenance
• Estimate maintenance costs
• Architectural Design Activities
• Design principles should emphasize modularity
• Make design simple to enable future enhancements
• Follow proper standards for all design diagrams, data flow, functions and documents
• Give high emphasis to information hiding, data abstraction and follow top-down design approach
Detailed Design Activities
• Follow proper standards for algorithms, data structures, variables declaration and so on
• Determine exceptions and error handling for the routines
• Provide cross-reference directories
• Implementation Activities
• Single entry – single exist constructs are the recommended methods to follow
• Use indentation
• Make coding simple and clear
• Make routines clear with appropriate comments and documents
Other Activities
• Create a maintenance guide to provide technical information and operating features of the system in detail
• Develop a Test Suite with the test cases which were developed during the implementation stages and during the testing phase.

- Provide associated documents for user manuals, test suites and system configurations.

Managerial Aspects of Software Maintenance

As with every branch of science and engineering, software engineering requires both technical skills as well as managerial skills. Project management is one of the toughest tasks to accomplish. Maintenance especially needs a lot of management skills in order to enhance product quality, easy maintainability and hence the customer satisfaction. It is always a trusted principle of expecting 'repeat-business' from established clients rather than looking for newer customers.

- Change Request Form
    - Software maintenance requires tracking and control of activities. Problem reports have to be tracked, recorded and solved accordingly.
    - This is accomplished using a systematic approach towards maintenance. When a change is necessary in the product, the activity occurs in response to a form filled by the end user of the product. This form is termed Change Request Form.
    - A change request may be feature enhancement, UI changes or an error fixation.
    - When a change request is made it is processed in the following systematic fashion.

Initiate software change request
Analyze request
Check for valid requests
If (not a valid request) then
    Close the request
Else
    Submit the request to the Change Control Board for analysis and acceptance.
    If (request approved by change control board) then
        Make changes in code based on requests with constraints specified by the change control board.
        Perform tests for the new changes and hence the system integrity.
        Submit the new changes to the change control board
        If (change control board approves the new changes) then
        Update external documentations and user manuals
        Make the changes known to everyone and the end users
        Else make changes as per the board's instructions and re-submit
Else Close the request
Endif

- The change request form is submitted by the user. It is reviewed by the analyst. If the analyst finds the problem is not an error or irrelevant to the software product then he discusses with the user and then close the request. If the problem is potentially huge then the form is analyzed, fixes proposed and resources estimated and the problem is brought to the notice of the change control board.

- Sometimes the problem might be a small error or a simple add-on to fixes or makes fresh changes to the code with the aid of maintenance programmers.
- **Change Control Board (CCB)**
    - The Change Control Board is a group of technical personnel who review the change requests from the users and propose fixes. Any change to the software product has to be first approved by the board before it is implemented.
    - Usually change control board refers to the software configuration control. However there are several other separate boards in larger projects such as Program CCB, Subsystem CCB, Software CCB and so on.
    - Once a change request is approved by the CCB then the problem is fixed by the maintenance programmers with the assistance from the analysts.
    - The composition of CCB and the hierarchy depends upon the kind of project in progress. For a large project the board may comprise several managers both from technical side as well as the management side and also senior business analysts who posses in-depth knowledge about the system.
    - The general members of a CCB maybe :
        - Quality Assurance Manager
        - Configuration Manager
        - Senior System Engineer
        - Test Engineers
    - The major advantage of solving user requests by the above method through the change request board is that the maintenance programmers are shielded from the end users and only the analysts interact with them.
- **Change Request Summaries**
    - As change requests are submitted and fixes provided, they are recorded and summaries or logs are generated.
    - The maintenance activities are summarized on weekly or monthly basis.
    - These reports provide details on the various change requests, fixes performed for these requests, emergency problems if any and so on.
    - The reports are done on a periodic basis which enable the management to keep in track of the versatility of the software product.

      The summaries also provide details on the number of open requests and the closed requests for each period during maintenance phase.
    - The summaries can also be used to generate TREND PLOTS to obtain a graphical representation of the maintenance activities.
    - Such a plot can be done to determine the number of open and closed requests and the frequency of bugs in the software products.
- **Quality Assurance activities**
    - The function of Quality Assurance Group (QA) is to ensure that the quality of software does not degrade during the maintenance period.

- As with development, QA plays an important part during maintenance phase also. Frequent audits, benchmarks and inspections are conducted to ensure the quality of the product as well as the documentations.
- The Software Configuration Management Plan (SCMP) describes the controls to be used to ensure the integrity in design, documentation and the final software. These controls provide guidelines for the establishment of baselines at major milestones during development as well as maintenance.
- The SCMP also specifies methods to be used in controlling changes to these baselines. Configuration management is vital to both development as well as maintenance activities.
- The QA team involves itself in the CCB activities. The change requests are brought to the notice of the QA team and once the fixes are provided the QA team performs regression testing for the modified software for standards and business – logic based on checklists and finally updates the master copy of the software and also the external documentation.

- **Organizing Maintenance Programmers**
  - Once implementation is over and the system is accepted by the customer, a set of programmers who have played their part in the development stages and who are aware of the under-lying architecture and the business – logics are allocated the task of maintaining the product. Such programmers are called Maintenance Programmers.
  - Maintenance can also be done by a separate group but it is always advantageous to maintain the product by the development team members since they are aware of the software product.
  - However, the use of the team members from the development team hampers their involvement into 'newer' projects and focuses attention on maintenance of the 'older' product.
  - On the other hand if maintenance is done by a separate group then the attention is more towards standards and high quality documentation.

    Though it takes some time to study the product, the effort of such a team into maintenance is always pristine.
  - Since there is a bit of trade-off in organizing maintenance teams, most projects try to rotate the programmers between development and maintenance. Since many developers feel maintenance as a field with not much of intellectual utility, the management usually motivate them by allowing them to develop their skills in both development and also maintenance. This enables them to learn new technologies, learn new skills from senor members of the team and greater flexibility in staffing.
  - Regarding of how maintenance is done, there is always a stigma associated with the word 'maintenance' and many programmers fear being allotted the 'dead work' of maintaining a product.

- However, it is a good method to involve at least two persons in maintenance. This provides a strong base for interaction and hence innovation in ideas. Also the programmers can inspect each other's work and the team does not always have to depend upon one programmer for the maintenance work.

Configuration Management

- Configuration management is the process of tracking and controlling the work products that constitute a software product.
- Configuration management generally consists of four general activities :

They are :

- ***Configuration Identification*** : selecting the appropriate documents that uniquely identify and define the configuration baseline attributes of an item.
- ***Configuration Control*** : controlling changes to the configuration and its identified documents.
- ***Configuration Auditing*** : checking an item for compliance with its configuration identification.
- ***Configuration Status Accounting*** : recording and reporting the implementation of changes to the configuration and its identification documents.
- During the course of development of a software project various milestones and audits are performed. When a work product passes a milestone review, then it is placed under configuration control and any subsequent control to this "accepted" product can be accomplished only with formal acceptance of the developing organization and the customer.
- Such a product that is under configuration control is termed "Base lined".
- ***Configuration Management Databases***

  This is a central repository which keeps in tracks of all activities throughout the entire software development life cycle of a particular software product. A Configuration Management Database provides information regarding product structure, the current revision number, current status and change request history for each version of the product. The following queries when raised to such a database should obtain the desired results
  - Versions of each product
  - Differences between versions
  - Available documents for each revised versions of each product.
  - The next revision date for a particular component of a given version.
  - Required hardware configuration for a specific version of a product.
  - Errors reported in a particular release.
  - Fixation of errors of a given version of the product.
  - Causes of product failures and causes of errors.
  - Functionally similar components among two given versions.
- ***Software Configuration Management Plan (SCMP)***

The Software Configuration Management Plan describes the controls to be used to ensure the integrity in design, documentation and the final software. These controls provide guidelines for the establishment of baselines at major milestones during the development cycle. The SCMP also specifies methods to be used in controlling changes to these baselines. Configuration management is vital to both development as well as maintenance activities.

- ***Version Control Libraries***

  A version control library system controls the various files that constitute the various versions of the software product. The version control system is not a database and does not have information to answer the queries as in Configuration Management databases. The version control libraries include
  - the actual source code,
  - the object code,
  - the data files
  - the supporting documents

  Since every entity of the product is under version control each of them is stamped with version number, date, time and its author.

  Example of a version control system is ***Revision Control System (RCS)***

  Another useful product is the Visual Source Safe Server. This supports graphical user interface with easy directory tree creations for the various files to be stored. When a developer wants to make changes in the code, the file has to be CHECKED OUT and hence becomes LOCKED. This prevents others editing the code and so prevents accidental overwriting of changes of the file. Once the developer is done with his changes, the file is CHECKED IN and is no more LOCKED and is said to be available.

SOURCE – CODE METRICS

- Metrics are the used to measure the complexity of source code.

Two source code metrics are discussed here:

- (1) Halstead's effort equation and (2) Meccabe's cyclomatic complexity measure

Halstead's Effort Equation

- Halstead developed a number of metrics from easily obtained properties of the source code. According to Halstead the basic parameters of a software are Operators and Operands.
- The properties used to determine the program metrics were
  - Total number of operators N1
  - Total number of operands N2
  - Number of unique operators in the code n1
  - Number of unique operands in the code n2

- Based on these properties Halstead defined many quantities using analogies to thermodynamics and from psychological studies.

Program Length : N = N1 + N2

N = n1log2n1 + n2log2n2

Program volume : V = (N1 + N2) log2 (n1 + n2)

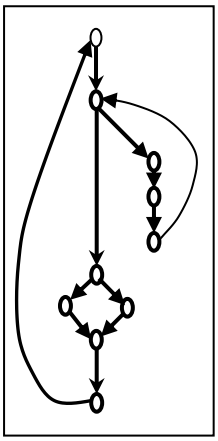
Level of Language abstraction : L = (2*n2) / (n1*N2)

Program effort : E = V/L

E = (n1*N2*(N1+N2)*log2(n1+n2)/(2*n2)

- **McCabe's Complexity Measure :**
    - McCabe observed the difficulty of a program is determined by the complexity of the control flow involved in the code.
    - The complexity number V of a connected G is the number of linearly independent paths in the graph.
      V(G)=E-n+2p

```
BEGIN
  SUM = 0
1 = 0
DO WHILE NOT EOF
  READ STUDENT FILE
  SUM = SUM + MARKS
END DO
IF (1>0) THEN
  AVE = SUM / 1
  PRINT AVE
ELSE
  PRINT "NO AVERAGE"
END IF
```



where

E is the number of edges

n is the number of nodes

p is the number of connected components

- Consider the following sample code for which a directed graph is drawn to find the Cyclomatic Number.
- In the above directed graph there are 12 edges, 10 nodes and since it is a planar graph p = 1. Hence we find the cyclomatic number as given by McCabe is
- V(G) = E+ n +p = 12 – 10 + 2(1) = 4

Maintenance Tools and Techniques

- Software maintenance as we have seen earlier is an important part of software engineering. Hence the role of a software maintainer is vital in the course of the product development and maintenance. A few tasks that are done by a maintainer are :
    - Negotiation with users
    - Analysis of change requests
    - Recommend solutions for the change control board

    - Involve in redesign, modification of software
    - Updating document based on changes done
    - Train users on the software product
    - Perform configuration control and provide temporary fixes
- In many organization, the software maintainer gets necessary inputs and assistance from business analysts as well as technical heads in performing the above mentioned tasks. Technical writers are handy in supporting the software maintainer by updating the required documents and materials.
- Apart from this, there are a few automated tools that assist the maintainer in performing his job more effectively and efficiently.
- There are two types of support tools
  Technical Support Tools and Managerial Support Tools.
- A few examples of tools assisting technical support for the maintainer are :
    - Text Editors
    - Debuggers
    - Cross – reference generators
    - Linkage editors
    - Comparators
    - Version control systems.

Text Editors

Text editors provide easy creation of programs, data and documents, edit them and save them for future purposes. There are several general purpose editors used for creating documents and programs Integrated Development Environment (IDE) is a feature which many popular languages support with options ranging from general text editing to rich text editing, search options, program execution and debugging, comment and indent code for much better readability and so on. Examples of text editors are VI, WORD, PAGEMAKER.

Debuggers

Debuggers provide methods to set breakpoints in a program and hence enable the user to test the part of code that is suspected to be improper or 'troublesome'. Debugging aids provide traps, memory dumps, traces and history files to aid the developers in detecting errors in code. Certain IDE programs provide in-built debuggers with 'Watch Windows' and 'Immediate Windows' to enable the user to find the value of a variable under consideration or test a condition. Just-In-Time debuggers are another useful debugging tools.

Cross-Reference generators

These provide cross-reference listings for procedures, functions, and data references. The details provided by these cross-reference generators are vital in many cases to detect, fix bugs and hence maintain the software. They typically provide information on local variables. Statement usages, who calls whom, no of times a routine is being called, function names etc.

Linkage editors

When there are several modules of compiled object code, a linker links them together and hence generates an executable code. This also forms an essential tool for the maintenance programmer in selectively compiling and then linking those modules alone into the software system. Certain routines in the software may be modified during the maintenance period. Such modules alone have to be selectively compiled and linked into the system. Here the linkage editors comes handy to the maintainer.

Comparator

This tool is used to compare two files and generate a listing of the differences between them. Comparators are used to compare different versions of a program or test case or documents. This provides details regarding what change in the code produced the desired result and also the adverse effects of a particular piece of code. An example of such a comparator is WINDIFF.

Version control system

Version control and configuration management systems are used to track the activities during the lifecycle of a software product. They keep track of versions of modules, releases of modules made on the respective dates, date and author of code. They also provide backups of master source code at regular intervals. The usage of version control systems is important in a large software project comprising of several developers who work on the controlling versions, preventing loss of code, keep history of code for easy testing as well as further enhancements of code and so on. An example of such a system is the Visual Source Sever.