Unit 1

Software Process

Software:

a) Instructions (Computer Programs) that when executed provide desired features, function, and performance.

b) Data structures that enable the programs to adequately manipulate information.

c) Documents that describe the operation and use of the programs.

Evolutionary Role of Software

Software performs 2 roles

1. **Product** – As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware.
2. **Vehicle/ Platform** – As a vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Functions provided by Software

1. It delivers most important product i.e. Information or Data.
2. It provides the Gateway for world wide information.
3. It provides the means of exchanging information.
4. It manages the business Information.

Software Applications (Categories of Computer Software)

Software applications run on different areas, they are as follows

1. System Software

System software is a collection of programs written to service other programs.

E.g. Compilers, Editors, File Management Utilities.

2. Real-time software

Software that monitors/ analyzes/ controls real-world events as they occur is called real time Software.

E.g. Real time manufacturing process control.

3. Business Software

Business information processing is the largest single software application area. They include software that accesses one or more large databases containing business information.

E.g. Payroll, Inventory, Accounts.

4. Engineering and scientific software

Engineering & Scientific software is the software required for the engineering & scientific development purpose.

It will include all **CASE** tools & System Stimulations.

E.g. LEX, YACC, CAD, CAM.

5. Embedded Software

Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets.

E.g. digital functions in an automobile such as fuel control, dashboard displays, and braking systems

6. Personal Computer Software

These soft wares are used for enhancing the personal computers & to facilitate more control to the user.

E.g. Computer Graphics, Multimedia, Text Editors.

7. Web Based Software

The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data.

8. Artificial Intelligence Software

Software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.

E.g. Expert Systems, Pattern Recognition.

Software Engineering

According to Firtz Bauer – Software engineering is defined as "The establishment and use of sound engineering principles in order to obtain the software that is economical, reliable and work efficiently on real machines".

According to Boehm – "Software engineering is the application of science and mathematics by which capabilities of computer equipments are made useful to man via computer programs, procedures and related documentation".

According to IEEE – Software engineering is the systematic approach to the development, operation, maintenance and retirement of the software.

"Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates".

Goals of Software Engineering

The primary goals of Software Engineering is

1. To improve the quality of software.
2. To increase productivity.
3. To increase the job satisfaction of software engineers.

Software engineering is based on the following disciplines

1. Computer Science.
2. Management Science.
3. Economics.
4. Communication Skills.
5. Engineering approach to problem solving.

Software Engineering is a labor intensive activity where high degree of communication is required among

1. Customers.
2. Managers.
3. Software Engineers.
4. Hardware Engineers.
5. Other related technologists.

Terms related to Software Engineering activities.

Size Factors

Project Size: - Project size is a major factor that determines the level of management control and the types of tools and techniques required on a software project. There are certain categories of projects depending on the size.

1) Trivial Projects:-

1. It involves one programmer often working part time.
2. Time spent is few days or few weeks.
3. The program will contain up to 500 lines of code.
4. The software would contain 10 to 20 subroutines
5. They are mostly personal software for exclusive use of the programmer.
6. Very little need for Formal analysis, elaborate design documentation or extensive planning.

2) Small Projects:-

1. Involves one programmer.
2. Duration of 1 to 6 months.
3. The program can contain 1000 to 2000 lines of code.
4. The total project may contain 25 to 50 routines.
5. They usually do not have interaction between other programs.
6. Generally used for small commercial applications including report generation and simple data manipulation.
7. There is negligible interaction between the programmer and customer.

3**) Medium size Projects:-**

1. They involve 2 to 5 programmers.
2. Time spent is between 1 to 2 years.
3. Lines of code range from 10,000 to 50,000 LOC.
4. Number of routines used ranges from 250 to 1000.
5. They include assemblers, compliers, process control systems, inventory systems, small MIS software etc.
6. It requires medium interaction between programmers and customers.
7. A certain degree of formality is required in Planning, Documentation and Project reviews.

4) Large Projects:-

1. Requires 5 to 20 programmers.
2. Project duration lies between 2 to 3 years.
3. It can consist of 50,000 to 100,000 LOC.
4. The entire project is a combination of several subsystems.
5. The modules in the project have significant interaction between other programs and software systems.
6. It requires multilevel management.
7. Examples are large compilers, small time sharing systems, graphics software for data acquisition and display, real time control systems etc.
8. These projects can involve extensive planning, analysis, risk estimation, standardized documentation, formal reviews etc.

5) Very Large Projects:-

1. It may require 100 to 1000 programmers.
2. Its duration may range from 4 to 5 years.
3. The number of software source instructions may reach 10 lakh LOC.
4. The entire project is divided in to several major subsystems that extensively interact with each other.
5. The subsystems are very complex and are developed separately under expert guidance.
6. The prominent feature of these kinds of projects involves real-time processing, telecommunication and multitasking.
7. Examples include large operating systems, Mega database systems, and Military command & control systems.

6) Extremely large projects:-

1. They employ 2000 to 5000 programmers.
2. Time period ranges nearly 10 years.
3. There are 1 million to 10 million LOC.
4. The entire project is an integration of several extremely large systems which further consist of several large subsystems.
5. These projects involve mainly real time processing, telecommunications, multitasking and distributed processing.
6. Examples include Air traffic control systems, Ballistic missile defense systems, Military command & control systems etc.

Quality and Productivity factors during Software Development

Following are the factors that affect the quality and productivity

1) Individual ability
2) Team communication
3) Product complexity
4) Appropriate notations
5) Systematic approaches
6) Controlling changes
7) Level of technology
8) Required reliability
9) Available time
10) Problem understanding
11) Stability of requirements
12) Required skills
13) Facilities and resources
14) Adequacy of training
15) Management skills
16) Appropriate goals
17) Rising expectations

1) Individual ability: - Production and maintenance of software is a labor intensive activity. So these factors are direct functions of individual ability and effort. There are two aspects to ability

a) The general competence of the individual and

b) Familiarity with the particular application area.

2) Team communication: - The new approach towards software development requires many individuals to be involved. Many recent innovations in software engineering, like design reviews, structured walkthroughs and code reading exercises improve communication between the programmers.

3) Product complexity: - There are three generally acknowledged levels of product complexity – a) Application programs, b) Utility programs and c) System level programs. Application programs have the highest productivity and Systems programs have the lowest productivity, measured in terms of LOC per programmer day.

4) Appropriate notations: - The representation schemes are of fundamental importance. Good and standardized notations can clarify the relationships and interactions, of any specific interest.

5) Systematic approach: - Following certain standard systematic approach in procedures and techniques makes the software development process very professional and contribute in improving the quality of the software product.

6) Controlling change: - The factors that contribute the factor of change in software are a) Design deficiencies in hardware,
b) Change in requirements due to poor understanding of problem,
c) External economic and political factors etc.

To minimize the change the following suggestions can be utilized
a) Use of appropriate notations and standard techniques,
b) Scope of the mechanism to change ie flexibility provided to the software,
c) During planning process the strategies for controlled changes should be included, etc.

7) Level of technology: - As the level of technology is changing rapidly hence modern programming languages should be used during development. Modern programming languages provide –

a) Improved facilities for data definition & data usage,

b) User-defined exception handling,

c) Concurrent programming etc.

Also varied tools are provided such as assemblers and primitive debugging facilities integrated under a single fully integrated development environment.

8) Level of reliability: - High reliability can be achieved by taking great care during –

a) Analysis,

b) Design,

c) System testing and

d) Maintenance.

Both human and machine resources are required to increase reliability. Too much reliability causes decreased productivity when measured in LOC produced Per Programmer Month.

9) Problem understanding: - Failure to understand the problem can result in an unsuccessful project. There are some suggestions for avoiding these problems –

a) Careful planning,

b) Customer interviews,

c) Task observation,

d) Prototyping,

e) Precise product specifications etc.

10) Available time: - Programmer productivity is sensitive to the, calendar time available for project completion. It is observed that the development time can not be compressed below 75 percent of the nominal development time. Extending a project beyond some nominal duration increases the total effort required.

11) Required Skills: - The practice of SE required variety of skills. The requirement definition and design activity are conceptual in nature hence require good creative problem solving skills. Debugging requires deductive or detective kind of skills. Preparations of external documents require good writing and expressive skills. Software Engineers should have good social skills to interact properly with the managers, customers and other engineers.

12) Facilities and resources: - Through various studies it had been found that work-related factors such as –

a) Good machine,

b) Quite place of work,

c) Plenty of access to the machine

d) Creative challenges,

e) Variety of tasks,

f) Opportunities for professional advancement etc, creates more satisfaction among the programmers rather than status factors.

13) Adequacy of training: - It is seen that the freshers from college do not have the following skills, which are very necessary for becoming a successful software engineer –

a) Express one self clearly in English

b) Develop & validate software requirements and design specifications.

c) Work with in application area.

d) Perform software maintenance.

e) Perform economic analysis.

f) Work with project management techniques.

g) Work in groups.

Hence adequate of training is required for entry-level programmers.

14) Management Skills: - As the concept of SE is new hence the managers who work in traditional styles are unable to properly manage the SE activities. Also it is seen that during giving promotions if the competency of the candidate is only measured in the technical point of view then there may arise the same kind of problem. So the management skills in a candidate should also include technical and managerial competency.

15) Appropriate goals: - Setting of appropriate goals is a major contributing factor in successful execution of a project. Generally the most common goals include

a) Generality,

b) Efficiency and

c) Reliability.

High productivity and quality factors can be achieved by adhering to the goals and requirements established for the software during project planning.

16) Rising expectations: - Progress is constantly being made in the development of tools and techniques to improve software quality and programmer productivity, equally diversity, size and complexity of software applications are growing at a fast rate but also the expectations and increasing demands. Hence it's necessary to adhere to the latest that is happening.

17) Stability of requirements: - The requirements of a software project includes

a) Manpower,

b) Software and

c) Hardware.

The requirements definition is done during problem analysis phase where the amount and type of resources are predicted. In this point if the analysis and prediction is not based on facts, expert guidance and experience from previous projects, then the requirements would change through out the development process and may extend to maintenance phase. Un-stability of requirements increases the cost, degrades overall quality and reduces the productivity. Hence great care should be taken during the initial phase of problem identification & problem analysis.

Other factors: - There are several other factors that influence the productivity –

a) Familiarity to the programming environment,

b) Amount and type of access to the computing system,

c) Stability of the computing system,

d) Memory & timing constrains,

e) Experience in that particular area of interest,

f) Data-base size etc.

Managerial Issues

Ideal or expected activities of Managers in a firm –

1. Managers should control the resources and the environment in which technical activities occurs.
2. They also have ultimate responsibility for ensuring that software products are delivered on time and with in cost estimates.
3. They have the responsibility of ensuring that the software products exhibit the functional and quality attributes desired by the customer.
4. The managers are responsible for project management which includes methods for organizing and monitoring the project progress.
5. Organizing and monitoring a project includes –

a) Cost estimation,

b) Resource allocation policies,

d) Budgetary control,

e) Setting project milestones,

f) Making schedule adjustments,

g) Establishing quality assurance procedures,

h) Establishing effective communication between project members,

i) Customer communications,

j) Development of contractual agreements with customers taking in to view legal obligations and responsibilities.

Software Life Cycle Models (Software Engineering Paradigm)

1. **Water Fall Model (Linear Sequential Model)**

1) It is a simplest model, which states that the phases are organized in a linear order.
2) The model was originally proposed by Royce.
3) The various phases in this model are

a) **Feasibility Study** – The main aim of the feasibility study activity is to determine whether it would be financially and technically feasible to develop the product. The feasibility study activity involves the analysis of the problem and collection of all relevant information relating to the product such as the different data items which would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system as well as various constraints on the behavior of the system.

b) **Requirement Analysis** – The aim of the requirement analysis is to understand the exact requirements of the customer and to document them properly. The requirement analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. During this activity, the user requirements are systematically organized into a software requirement specification (SRS) document. The important components of this document are the functional requirements, the nonfunctional requirements, and the goals of implementation. The SRS document is written in end-user terminology. This makes the SRS document understandable by the customer. After all, it is important that the SRS document be reviewed and approved by the customer.

c) **Design** – This phase is concerned with

1) Identifying software components like functions, data streams and data stores.
2) Specifying software structure.
3) Maintaining a record of design decisions and providing blue prints for the implementation phase
4) The are two categories of Design

a) **Architectural Design** – It involves

1. Identifying the software components.
2. Decoupling and decomposing the software components into modules and conceptual data structures.
3. Specifying the interconnection between the various components.

b) **Detailed Design** – It is concerned with details of the implementation procedures to process the algorithms, data structures and interaction between the modules and data structures. The various activities that this phase includes are

1. Adaptation of existing code.
2. Modification of existing algorithms.
3. Design of data representation and
4. Packaging of the software product.

This process is highly influenced by the programming language under the implementation would be carried out. This stage is different from implementation.

d) **Implementation** – It involves the translation of the design specifications into source code. It also involves activities like debugging, documentation and unit testing of the source code. In this stage various styles of programming can be followed like built-in and user defined data types, secure type checking, flexible scope rules, exception handling, concurrency control etc.

e) **Testing** – It involves two kinds of activities

1. **Integration Testing** – It involves testing of integrating various modules and testing there overall performance due to their integration.
2. **Acceptance Testing** – It involves planning and execution of various types of tests in order to demonstrate that the implemented software system satisfies the requirements stated in the requirements document.

f) **Maintenance** – In this phase the activities include

1. **Corrective Maintenance** – Correcting errors that were not discovered during the product development phase.
2. **Perfective Maintenance** – Improving the implementation of the system, and enhancing the functionalities of the system according to customer's requirements.
3. **Adaptive Maintenance** – Adaptation of software to new processing environments.

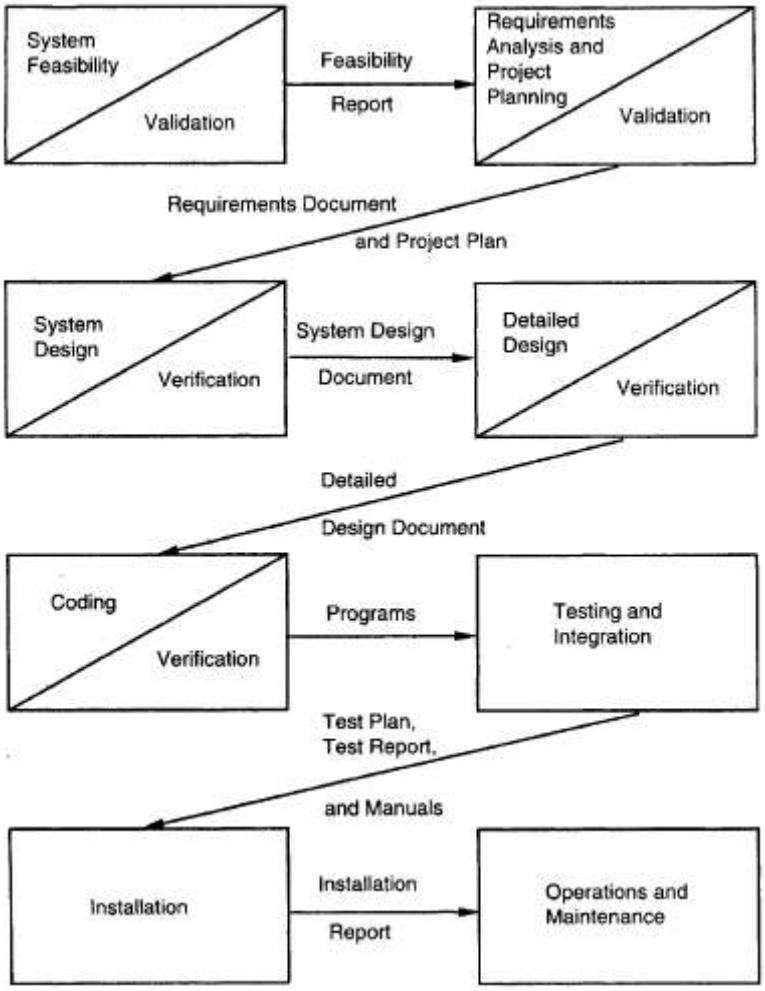


Fig Waterfall Model

Advantages of Water Fall model

1. All phases are clearly defined.
2. One of the most systematic methods for software development.
3. Being oldest, this is one of the time tested models.
4. It is simple and easy to use.

Disadvantages of Water Fall Model

1. Real Projects rarely follow sequential model.
2. It is often difficult for the customer to state all requirements explicitly.
3. Poor model for complex and object oriented projects.
4. Poor model for long and ongoing projects.
5. High amount of risk and uncertainty.
6. Poor model where requirements are at a moderate to high risk of changing.
7. This model is suited to automate the existing manual system for which all requirements are known before the design starts. But for new system, having such unchanging requirements is not possible.

2. Incremental Model

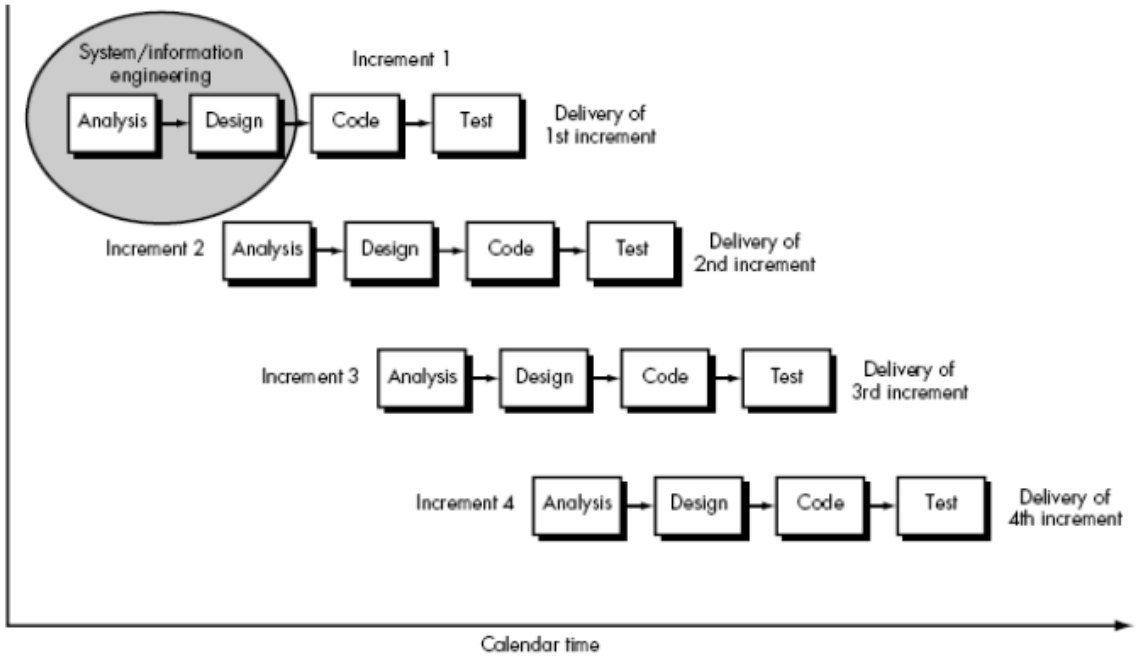
1. The ***incremental model*** combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping.
2. It is iterative in nature.
3. It focuses on the delivery of the operational product with each increment
4. The process is repeated until the complete product is produced.
5. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
6. When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered.
7. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
8. This process is repeated following the delivery of each increment, until the complete product is produced.
9. Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

Advantages

1. It provides on the rigid nature of sequential approach.
2. This method is of great help when organization is low on staffing.
3. Generates working software quickly and early during the software life cycle.
4. More flexible – less costly to change scope and requirements.
5. Easier to test and debug during a smaller iteration.
6. Easier to manage risk because risky pieces are identified and handled during its iteration.
7. Each iteration is an easily managed milestone.

Disadvantages

1. This model could be time consuming.
2. Each phase of an iteration is rigid and do not overlap each other.
3. Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.



3. <u>Spiral Model</u>

1. Evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model.
2. It provides the potential for rapid development of incremental versions of the software.
3. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.
4. A spiral model is divided into a number of framework activities, also called *task regions.*
5. There are six task regions.

   1. Customer communication—tasks required to establish effective communication between developer and customer.

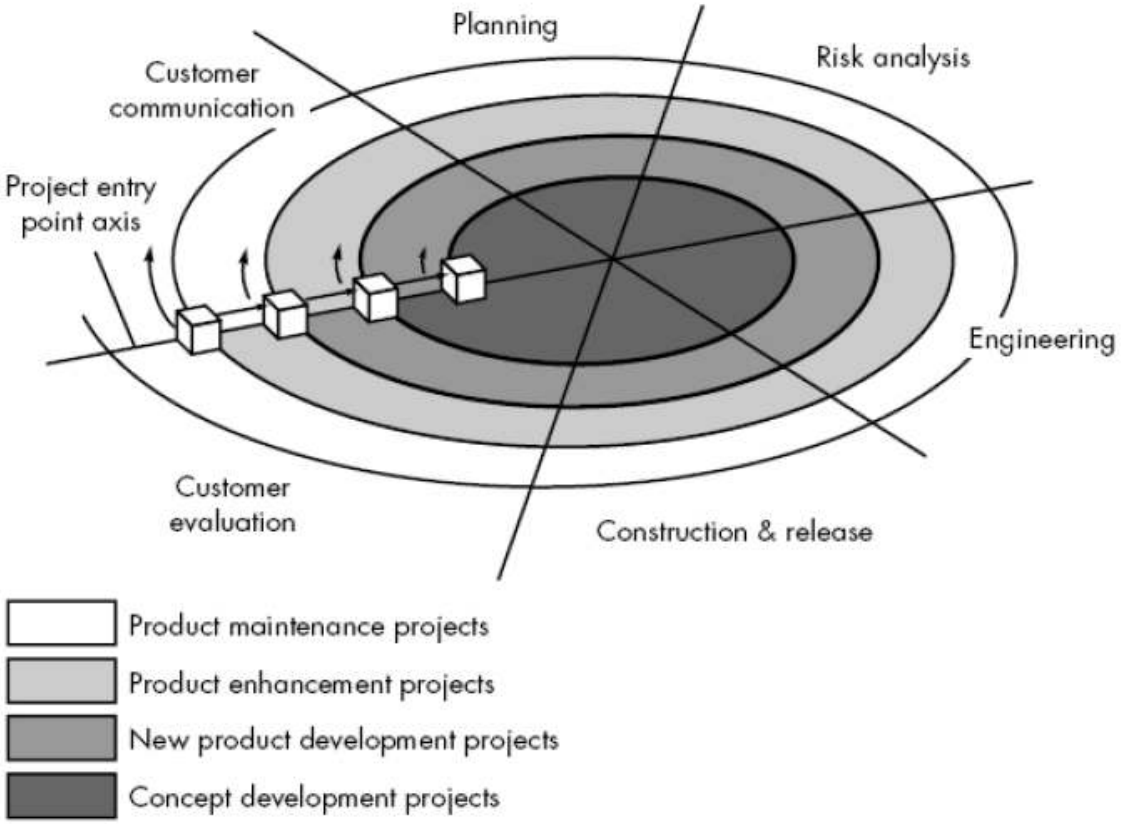
   2. Planning—tasks required to define resources, timelines, and other project related information.

   3. Risk analysis—tasks required to assess both technical and management risks.

   4. Engineering—tasks required to build one or more representations of the application.

   5. Construction and release—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

   6. Customer evaluation—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.



Advantages:

1. It is a realistic approach for development of large scale system.
2. High amount of risk analysis
3. Good for large and mission-critical projects.
4. Software is produced early in the software life cycle.

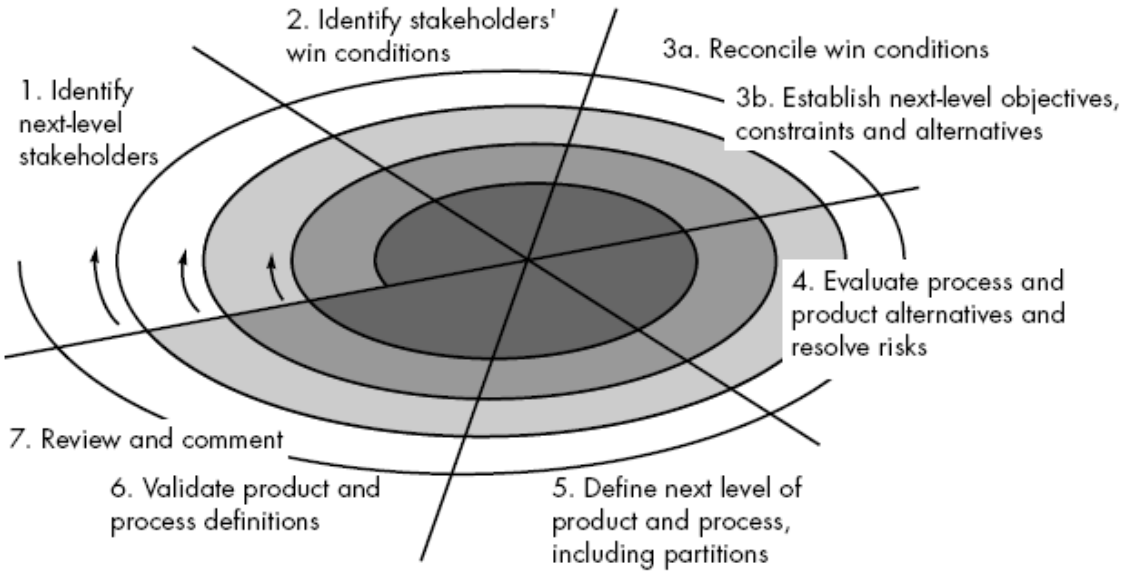
Disadvantages:

1. It is not widely used.
2. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
3. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.
4. Can be a costly model to use.
5. Risk an alysis requires highly specific expertise.
6. Project's success is highly dependent on the risk analysis phase.
7. Doesn't work well for smaller projects.

4. <u>WINWIN Spiral Model</u>

1. This is an adaptation of the spiral model which emphasis is explicitly placed on the involvement of the client in a negotiation process at the genesis of the product development. Ideally, the developer would simply ask the customer what is required and the customer would provide sufficient detail to proceed. Unfortunately this rarely happens and significant negotiations between both parties are required to balance functionality, performance, etc… with cost and time-to-market considerations.
2. First WIN is for customer and second one is for developer.
3. The best negotiations strive for a "WIN-WIN" result. That is, the customer wins by getting the system or product that satisfies the majority of the customer's needs and the developer wins by working to realistic and achievable budgets and deadlines.

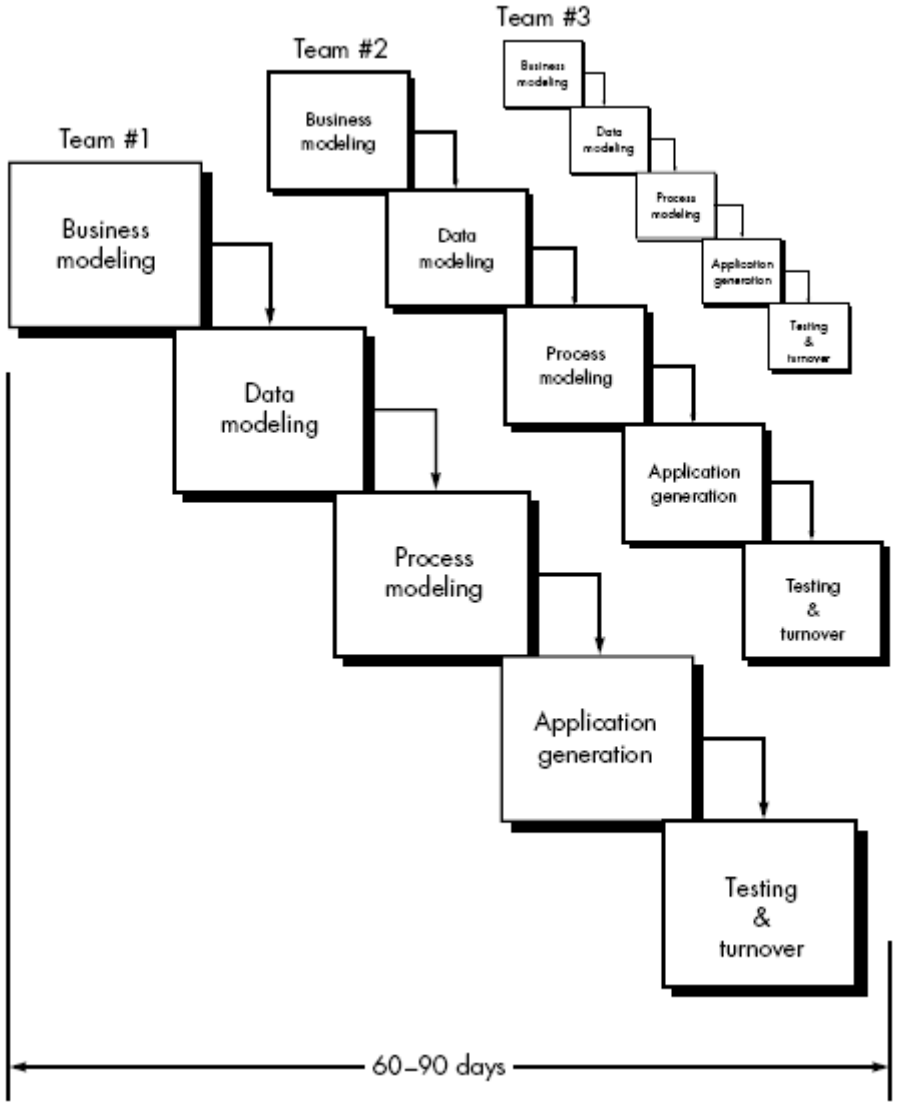
4. WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity, the following activities are defined.
    a) Identification of the system or subsystem's key "Stake holders".
    b) Determinations of the Stake holder's "WIN conditions".
    c) Negotiations of the Stake holder's WIN conditions to reconcile them into a set of "WIN-WIN" conditions for all concerned (including the software project team).
5. In addition to the emphasis placed on early negotiations, the WINWIN spiral model introduces three process milestones, called anchor points that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.
6. In essence, the anchor points represent three different views of progress as the project traverses the spiral.
7. The first anchor point, life cycle objectives (LCO), defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establish the definition of top-level system/product requirements.
8. The second anchor point, life cycle architecture (LCA), establishes objectives that must be met as the system and software architecture is defined. For example, as part of LCA, the software project team must demonstrate that it has evaluated the applicability of off-the-shelf and reusable software components and considered their impact on architectural decisions.
9. Initial operational capability (IOC) is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation and assistance required by all parties that will use or support the software.



Advantages

1. Flexibility – The model let the teams adapt to accompanying risks and uncertainties, such as a rapid project schedule and changing team composition.
2. Discipline – The modeling framework was sufficiently formal to maintain focus on achieving three main, or "anchor-point", milestones: The life-cycle objectives, the life cycle architecture, and the initial operational capability.
3. Trust Enhancement – The model provided a means for growing trust among the project stakeholders, enabling them to evolve from adversarial, contact-oriented system development approaches.

5. RAD Model



1. Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle.
2. The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction.
3. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days).
4. Used primarily for information systems applications, the RAD approach encompasses the following phases
    a) **Business Modeling: -** This phase involves identification of various information flows between various business functions that act as the driving force in the development process. It's a kind of problem identification task.
    b) **Data Modeling: -** The relationships between various data elements are identified. It's like the planning phase where the data which drives the business functions are identified and planned out to establish synchronization so that the designing could be done.
    c) **Process Modeling: -** It's like the design phase. Here the properly modeled data are put together such that actual coding to implement the functionalities for data manipulation could be done.

d) **Application generation: -** It's like the coding and implementation phase. Here the planned out processes are actually coded and executed. In this phase it's taken in to consideration that the code should be made is such a manner that they become reusable.

e) **Testing and turnover: -** Here the final code is repeatedly tested for reliability, robustness and maintainability. If some short comings are encountered then again the initial stages are repeated for identification and eradication of errors.

5. In this type of development process a problem is initially analyzed in details such that the problem could be divided or decomposed into manageable smaller problems. The analysis is done in such a way that the decomposed problem could be solved parallel and the solutions could be integrated to solve the entire problem.

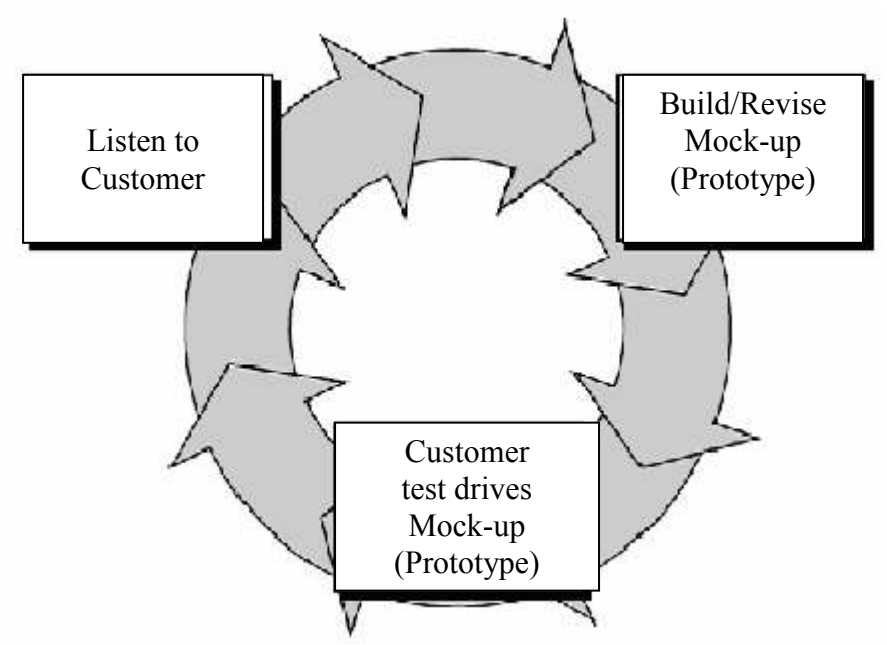
Advantages

1. As RAD stresses on reuse of components testing time may be very less in certain cases.
2. RAD model can be most efficient when it comes to overall development time.

Disadvantages

1. For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
2. RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail.
3. Not all types of applications are appropriate for RAD. If a system cannot be properly modularized, building the components necessary for RAD will be problematic. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
4. RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

6. Prototyping Model



1. Often, a customer defines a set of general objectives for software, but does 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 form that human-machine interaction should take. In these and many other situations, a prototyping paradigm may offer the best approach.

2. The various phases of this model are
    a) **Listen to Customer: -** This is the starting step, where the developer and customer together
        - Define the overall objectives for the software,
        - Identify the known requirements and
        - The analyst then outlines those factors and requirements that are not visible normally but are mandatory from development point of view.
    b) **Build prototype: -** After the identification of the problem a quick design is done which will cause the software show the output that the customer wants. This quick design leads to the development of a prototype (a temporary working model).
    c) **Customer test drives the prototype: -** The customer runs and checks the prototype for its perfection. The prototype is evaluated by the customer and further improvements are made to the prototype unless the customer is satisfied.
3. All the stages are repeated until the customer gets satisfied. When the final prototype is fully accepted by the customer then final development processes like proper coding for attaining maintainability, proper documentation, testing for robustness etc. are carried out. And finally the software is delivered to the customer.

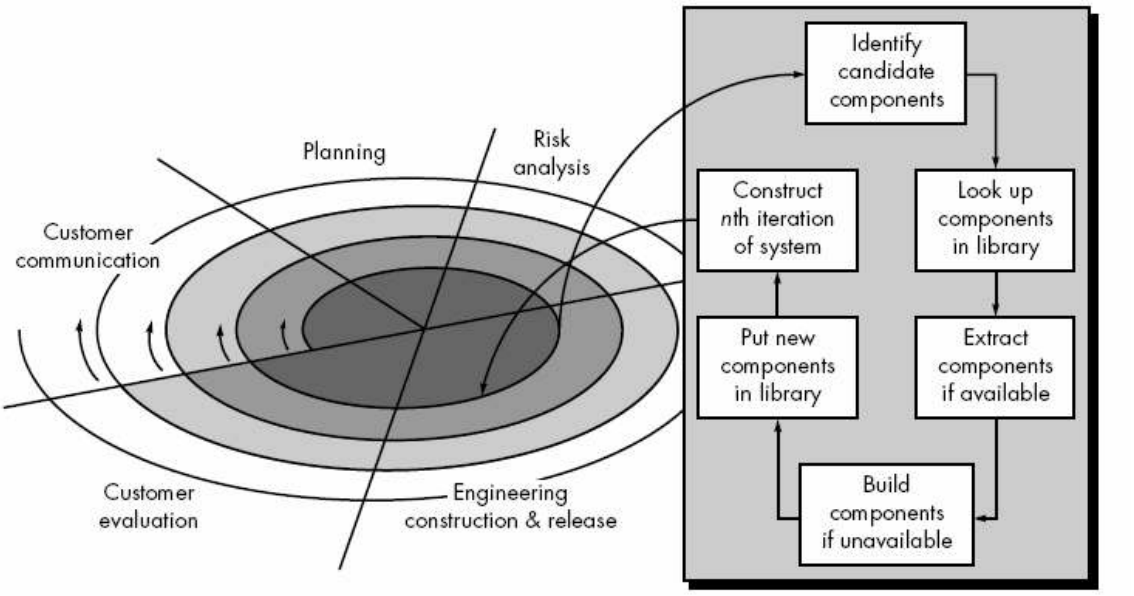
Advantages

1. It serves as the mechanism for identifying the requirements.
2. The developer use the existing program fragment means it helps to generate the software quickly.
3. Continuous developer – Consumer communication is available.

Disadvantages

1. Customer considers the prototype as an original working version of the software.
2. Developer makes implementation compromise in order to get prototype working quickly.
3. This model is time consuming.

7. Object Oriented Model (Component Assembly Model)



1) This kind of model is used for object based (oriented) development process.

2) Here the emphasis is on the creation of reusable classes that encapsulate both data and functions, where the functions can to manipulate the data.
3) This model incorporates many of the characteristics of spiral model and iterative approach towards software development.
4) During the Engineering phase and Construction & release phase, the data is examined and accordingly algorithms applied for data manipulation is also decided. Corresponding data and algorithms are encapsulated into a Class.
5) Classes created in past are stored in a class library. Once the required classes are identified, the classes libraries are searched, if they are found then they are reused. In case where the required classes are not found in the library then they are engineered using object oriented methods.
6) When the process of recognition and usage of classes is done then the development process returns back to the spiral path and subsequently if required reenter in the component assembly process during successive iterations, unless fully acceptable product is made.

Difference between WaterFall model and Spiral model

No.	WaterFall Model	Spiral Model
1	It is a Systematic, sequential approach to software development that begins at system level and progress through analysis, design, coding, testing, and maintenance.	In this model software is developed in a series of incremental versions by the process of iteration through a number of framework activities like Customer communication, planning, risk analysis, engineering, construction engineers and customer evaluation.
2	It is the oldest and the most widely used paradigm in software engineering.	It is an evolutionary approach to software engineering and is currently gaining foothold.
3	It does not include provisions for risk assessment.	Risk assessment is one of the major activities of the spiral model.
4	It is not convenient to model real life large scale projects since most of them do not follow a sequential flow that the model proposes.	Spiral model incorporates the steps of the waterfall model in an interactive framework and hence can more realistically reflect the real world system.
5	This model requires the specification of all requirements at a very early stage in the development, which is often difficult to do.	The Spiral model iterates through the stages of development several times, each time providing a more improved version of the previous stage, additional requirements can be incorporated at later stages of development.
6	The effects of major risks taken can be gauges only at a very later stage of development and by then it would be too terminating the project.	Spiral model requires considerations of risks at all stages of development and if risk are too great, the process can be terminated at a much earlier stage.
7	A working version of the program is available at a much later stage of the project.	A skeleton working model of the system can be developed at earlier stage, which is the refined in successive interaction.
8	No planning is done.	Every pass through planning results in the adjustment of large scale systems of the customer.
9	Entry point is not present.	Entry points are specified.

Q. Explain why the waterfall model of the software process is not an accurate reflection of the software development activities.

Ans – The waterfall model of the software process is not accurate because

1. The linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete development tasks.
2. In fact, the time spent waiting can exceed the time spent on procedure work. The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.
3. Each of these problems is real.
4. However, the classic life cycle paradigm has a definite and important place in software engineering work.
5. It provides a template into which methods for analysis, design, coding, testing, and support can be placed.
6. The classic life cycle remains a widely used procedural model for software engineering.
7. While it does have weakness, it is significantly better than a haphazard approach to software development.
8. The waterfall approach assumes that a complete system will be delivered after the linear sequence is completed.
9. The evolutionary nature of software is not considered in either of these classic software engineering paradigms.

Q. Which of the development process models would you follows for the following projects. Give justifications.

a) A simple data processing project.

Ans: For above project I would like to go for **waterfall model.**

Justification

In the given project, the requirement will be fixed and there is no near chance of changing it. Also for processing data, those basic operations will be fixed and we have delivered all the operations at a time.

In waterfall model, we can very well implement all specified requirements and deliver the whole product at a time.

b) A new system comparing finger prints.

Ans: For above project I would like to go for **spiral model**.

Justification

In the given project, requirement i.e. finger print data will be added continuously so we need to consider it. Even customer feedback i.e. usefulness of the project has to be checked every time. In spiral model, for each set of requirement we can follow up six activities and in turn that spiral way will be continuing till the project is delivered.

c) An online inventory management system for an automobile industry.

Ans: For above project I would like to go for **incremental model**.

Justification

In this project, we will be delivering new services in every increment. Hence, in online MIS system at a time, it is not possible to deliver the entire module and we can not, even delay the delivery. So for this reason, we can consider some features and deliver first increment. Later on, in each increment we can add new features and deliver the entire module increment by increment.

d) A new missile tracking system.

Ans: For this project I would like to go for **waterfall model**.

Justification

Here for missile tracking system the frequency range for tracking missiles will be fixed. All data will be provided at start only. Using this data the entire system has to deliver at a time only so waterfall model is useful here.

e) A satellite launching system. It is not known if the current H/W and S/W technology is nature enough to achieve the goals.

Ans: For this project I would like to go for **prototype model**.

Justification

Satellite launching system is very costly system and current hardware and software are not matching. The basic idea here is that instead of freezing is the requirement before any design or coding can proceed. Throwaway prototype is build to help understand the requirements. Development of the prototype obviously undergoes design, coding and testing but each of these phases is not done very formally or thoroughly.

f) An on-line inventory management for peripheral device manufacturing company.

Ans: For this project I would like to go for **waterfall model**.

Justification

This software is not so much costly. The software team gets all the information from the user and then analysis will be frozen. After the analysis frozen system has developed.

g) A data entry system for office staff that have never used computers before. The user interface and user friendliness are extremely important.

Ans: for this project I would like to go for **increment model**.

Justification

This software is also not so much costly. The basic idea is such that the software is developed in increment, each increment adding some functional capability to the system until the system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier then testing the entire system as in the waterfall model.

Software Development Process

In the software development process, we have to focus on the activities directly related to production of the software, for example, design, coding, and testing. A development process model specifies some activities that, according to the model, should be performed, and the order in which they should be performed. For cost, quality, and project management reasons, development processes are generally phased.

As the development process specifies the major development and quality assurance activities that need to be performed in the project, the development process really forms the core of the software process. The management process is decided, based on the development process. Due to the importance of development process, various models have been proposed. As processes consist of a sequence of steps, let us first discuss what should be specified for a step.

A Process Step Specification

A production process is a sequence of steps. Each step performs a well-defined activity leading towards the satisfaction of the project goals, with the output of one step forming the input of the next one. Most process models specify the steps that need to be performed and the order in which they need to be performed. However, when implementing a process model, there are some practical issues, like when to initiate a step and when to terminate a step, that need to be addressed. Here we discuss some of these issues.

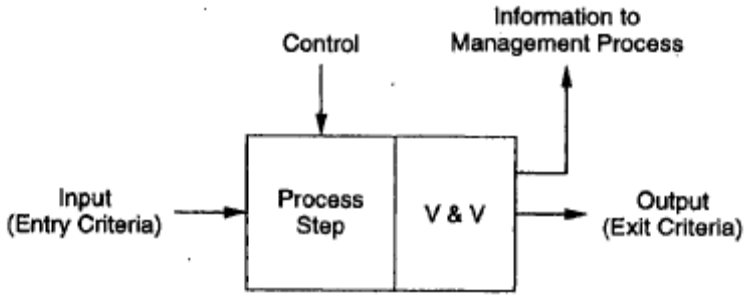
A process should aim to detect defects in the phase in which they are introduced. This requires that there be some verification and validation (V & V) at the end of each step. (In verification, consistency with the inputs of the phase is checked, while in validation the consistency with the needs of user is checked.) This implies that there is a clearly defined output of a phase, which can be verified by some means and can form input to the next phase (which may be performed by other people). In other words, it is not acceptable to say that the output of a phase is an idea or a thought in the mind of someone; the output must be a formal and tangible entity. Such outputs of a development process,

which are not the final output, are frequently called the work products. In software, a work product can be the requirements document, design document, code, prototype, etc.

This restriction that the output of each step should be some work product, that can be verified, suggests that the process should have a small number of steps. Having too many steps results in too many work products or documents, each requiring V & V, and can be very expensive. Due to this, at the top level, a development process typically consists of a few steps, each satisfying a clear objective and producing a document used for V & V. How to perform the activity of the particular step or phase is generally not an issue of the development process.

As a development process, typically, contains a sequence of steps, the next issue that comes is when a phase should be initiated and terminated. This is frequently done by specifying the entry criteria and exit criteria for a phase. The entry criteria of a phase specify the conditions that the input to the phase should satisfy in order to initiate the activities of that phase. The output criteria specify the conditions that the work product of this phase should satisfy in order to terminate the activities of the phase. The entry and exit criteria specify constraints of when to start and stop an activity. It should be clear that the entry criteria of a phase should be consistent with the exit criteria of the previous phase.

The entry and exit criteria for a phase in a process depend largely on the implementation of the process. For example, for the same process, one organization may have the entry criteria for the design phase as "requirements document signed by the client" and another may have "no more than X errors detected per page in the requirement review." As each phase ends with some V & V activity, a common exit criteria for a phase is "V & V of the phase completed satisfactorily," where satisfactorily is defined by the organization based on the objectives of the project and its experience in using the process. The specification of a step with its input, output, and entry exit criteria is shown in Figure below.



A step in the development process

Verification & Validation (V&V)

The goals of verification and validation activities are to asses and improve the quality of the work products generated during development and modification of software. The quality attributes include – a) Correctness, b) Completeness, c) Consistency, d) Reliability, e) Usefulness, f) Conformance to standards and g) overall cost effectiveness.

Verification: - It's the process of determining whether the product is built in a right manner or not. There are two categories of verification:

a) Life-cycle verification and

b) Formal verification.

a) Life-Cycle verification: - It's the process of determining the degree to which the work products of a given phase of the development cycle fulfill the specifications established during prior phases.

b) Formal verification: - It's a rigorous mathematical demonstration that whether the source code conforms to the specification.

Validation: - It's the process of evaluation of the software at the end of the software development process to determine compliance with the requirements. In other words it's the process of determining whether the correct product is built or not.

Verification and validation involve the assessment of work products to determine conformance to the specifications. Specifications include –

a) Requirements specification
b) The design documentation
c) Various stylistic guidelines
d) Implementation language standards
e) Project standards
f) Organizational standards
g) User expectations.

Verification

1. Verification is done to ensure that the work product of a given phase of the development cycle fulfill the specifications established during prior phase.
2. According to Boehm :

   Verification: Are we producing the product right?
3. Verification ensures the product is designed to deliver all functionality to the customer; it typically involves reviews and meetings to evaluate documents, plans, code, requirements and specifications; this can be done with checklists, issues lists, and walkthroughs and inspection meetings.
4. In verification uncovering of defects will be done in primary ways. Here no code will be executed. Before building actual system this checking will be done. This process will be called Quality Assurance.
5. Verification is nothing but the Static Testing.
6. Inputs to the verification are check list, issue list, walkthroughs and inspection meetings, reviews and meetings.
7. The output of the verification is a nearly a perfect set of documents, plans, specifications and requirements document.
8. According to the CMM, Validation - The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

Validation

1. Validation is the process of evaluating software at the end of the software development process to determine compliance with the requirements.
2. According to Boehm:

   Validation: Are we producing the right product?
3. Validation ensures that functionality, as defined in requirements, is the intended behavior of the product; validation typically involves actual testing and takes place after verifications are completed.
4. Validation concern, checking will be done by executing code for errors (defects.). This can be called as Quality Control.
5. Validation is nothing but the Dynamic Testing.
6. The input on the validation on the other hand is the actual testing of an actual product.
7. The output of the validation on the other is a nearly perfect, actual product.
8. According to CMM, Verification- The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Difference between Verification and Validation

No.	Verification	Validation
1	Verification is done to ensure that the work product of a given phase of the development cycle fulfill the specifications established during prior phase.	Validation is the process of evaluating software at the end of the software development process to determine compliance with the requirements.
2	Verification: Are we producing the product right?	Validation: Are we producing the right product?
3	Verification ensures the product is designed to deliver all functionality to the customer; it typically involves reviews and meetings to evaluate documents, plans, code, requirements and specifications; this can be done with checklists, issues lists, and walkthroughs and inspection meetings.	Validation ensures that functionality, as defined in requirements, is the intended behavior of the product; validation typically involves actual testing and takes place after verifications are completed.
4	In verification uncovering of defects will be done in primary ways. Here no code will be executed. Before building actual system this checking will be done. This process will be called Quality Assurance.	Validation concern, checking will be done by executing code for errors (defects). This also can be called as Quality Control.
5	Verification is nothing but the Static Testing.	Validation is nothing but the Dynamic Testing.
6	Verification is done before validation.	Validation cannot be done before verification.
7	Inputs to the verification are check list, issue list, walkthroughs and inspection meetings, reviews and meetings.	The input on the validation on the other hand is the actual testing of an actual product.
8	The output of the verification is a nearly a perfect set of documents, plans, specifications and requirements document.	The output of the validation on the other is a nearly perfect, actual product.

System Engineering

System

1. A set or arrangement of things so related as to form a unity or organic whole.
2. A set of facts, principles, rules, etc., classified and arranged in an orderly form so as to show a logical plan linking the various parts.
3. A method or plan of classification or arrangement.
4. An established way of doing something, method, procedure….

Computer Based System

"A set of arrangements of elements that are organized to accomplish some predefined goal by processing information."

The goal may be to support some business function or to develop a product that can be said to generate business revenue. To accomplish the goal, a computer based system makes use of a variety of system elements:

Software – Computer programs, data structures, and related work products that serve to affect the logical method, procedure, or control that is required.

Hardware – Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.

People – Users and operators of hardware and software.

Database – A large, organized collection of information that is accessed via software and persists over time.

Documentation – Descriptive information (e.g., models, specifications, hardcopy manuals, online help files, web sites) that portrays the use and/or operation of the system.

Procedures – The steps that define the specific use of each system element or the procedural context in which the system resides.

These elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and control software to support the robot both require system engineering.

System Engineering

"System engineering is an activity of specifying, designing, implementing, validating, deploying and maintaining technical system."

System engineers are not just concerned with software but also with hardware and the system's interactions with users and its environment. They must think about the services that the system provides the constraints under which the system must be built and operated and the ways in which the system is used to fulfill its purpose.

Difference between system Engineering and software Engineering

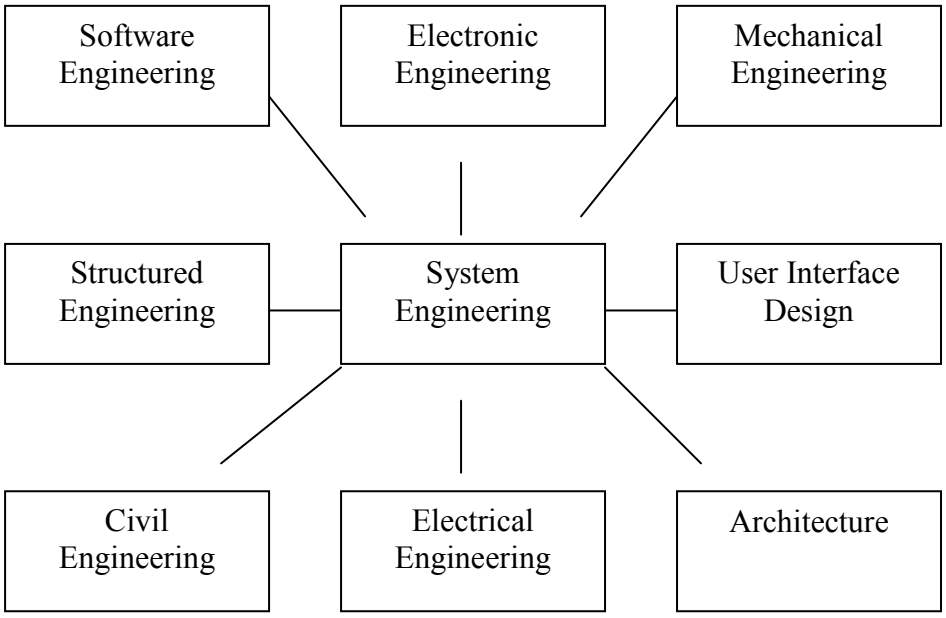


Fig Disciplines involved in systems engineering

There are important distinctions between the system engineering process and software development process

1. **Limited scope of rework during system development** – Once some system engineering decisions have been made, they are very expensive to change. Reworking the system design to solve these problems is rarely possible. One reason software has become so important in systems is that it allows changes to be made during system development, in response to new requirements.
2. **Interdisciplinary involvement** – Many engineering disciplines may be involved in system engineering. There is a lot of scope for misunderstanding because different engineers terminology and conventions.

System engineering hierarchy

System engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated in fig (a). The system engineering process usually begins with a "world view". That is, the entire business or product domain is examined to ensure that the proper business or technology context can be established. The world view is refined to focus more fully on a specific domain of interest. Within a specific domain, the need for targeted system elements (e.g., data, software, hardware, and people) is analyzed. Finally, the analysis, design, and construction of a targeted system element is initiated. At the top of the hierarchy, very broad contexts are established and, at the bottom, detailed technical activities, performed by the relevant engineering discipline (e.g., hardware or software engineering), are conducted.

Stated in a slightly more formal manner, the world view (WV) is composed of a set of domain (D_j), which can each be a system or system of systems in its own right.

$$WV = \{D1, D2, D3 \ldots Dn)$$

Each domain is composed of specific elements (Ej) each of which serves some role in accomplishing the objective and goals of the domain or component:

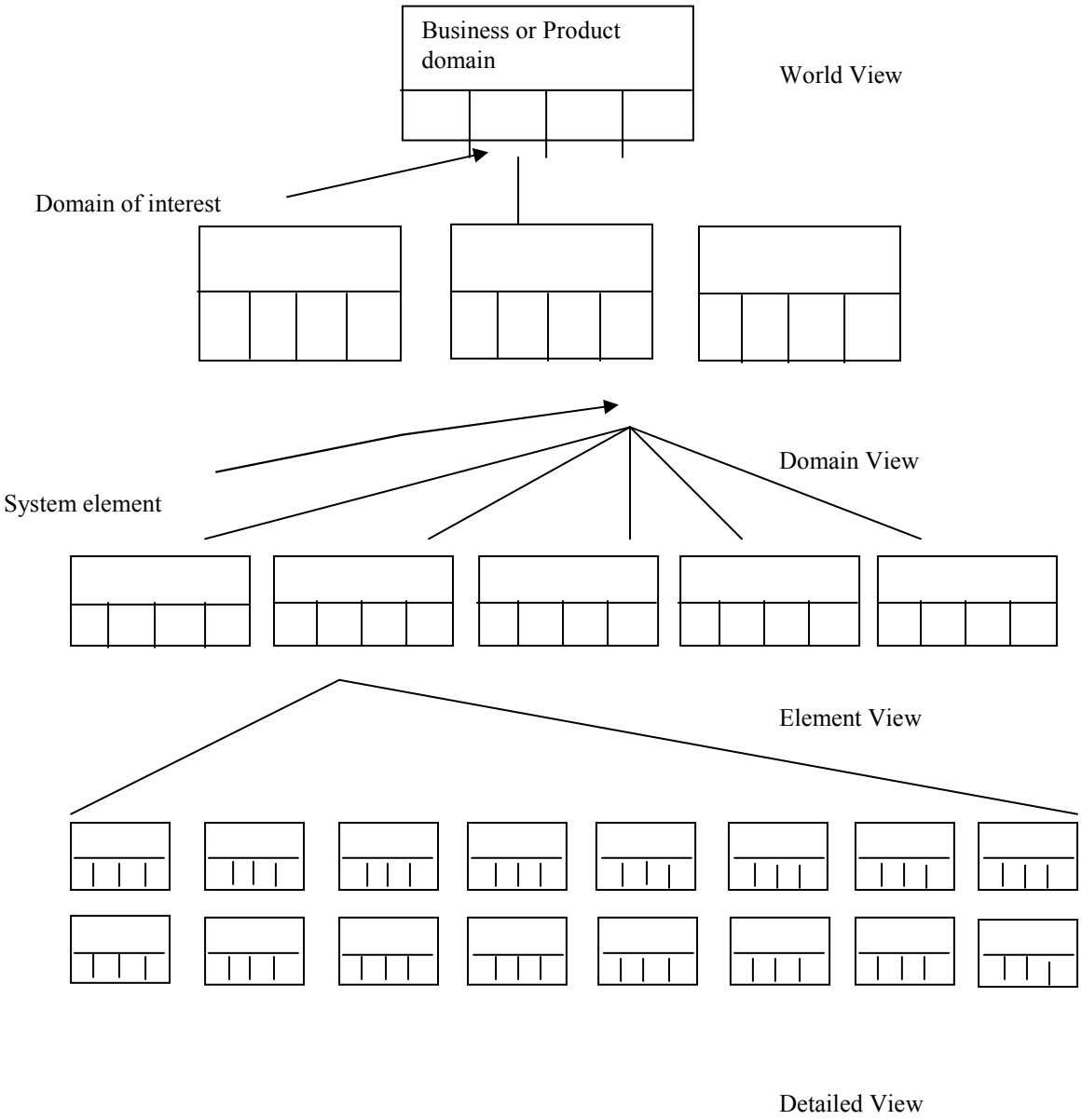
$$Di = \{E1, E2, E3 \ldots En\}$$

Finally, each element is implemented by specifying the technical components (Ck) that achieve the necessary function for an element:

$$Ej = \{C1, C2, C3 \ldots Cn\}$$

In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

It is important to note that the system engineer narrows the focus of work as she moves downward in the hierarchy. However, the world view portrays a clear definition of overall functionality that will enable the engineer to understand the domain, and ultimately the system or product, in the proper context.



Fig(a) System Engineering Hierarchy

Q. Explain the approach used by SEI (Software Engineering Institute) to determine the current state of process maturity of an organization.

Ans: The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1: Initial.

The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable.

Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined.

The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

Level 4: Managed.

Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing.

Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

The SEI has associated key process areas (KPAs) with each of the maturity levels. The KPAs describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics:

1. ***Goals***—the overall objectives that the KPA must achieve.
2. ***Commitments***—requirements (imposed on the organization) that must be met to achieve the goals or provide proof of intent to comply with the goals.
3. ***Abilities***—those things that must be in place (organizationally and technically) to enable the organization to meet the commitments.
4. ***Activities***—the specific tasks required to achieve the KPA function.
5. ***Methods for monitoring implementation***—the manner in which the activities are monitored as they are put into place.
6. ***Methods for verifying implementation***—the manner in which proper practice for the KPA can be verified.

Eighteen KPAs (each described using these characteristics) are defined across the maturity model and mapped into different levels of process maturity. The following KPAs should be achieved at each process maturity level:

1. **Process maturity level 2**
    a) Software configuration management
    b) Software quality assurance
    c) Software subcontract management
    d) Software project tracking and oversight
    e) Software project planning
    f) Requirements management
2. **Process maturity level 3**
    a) Peer reviews
    b) Inter – group coordination
    c) Software product engineering
    d) Integrated software management
    e) Training program
    f) Organization process definition
    g) Organization process focus

3. **Process maturity level 4**
    a) Software quality management
    b) Quantitative process management

4. **Process maturity level 5**
    a) Process change management
    b) Technology change management
    c) Defect prevention

Each of the KPAs is defined by a set of ***key practices*** that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted. The SEI defines ***key indicators*** as "those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved."

Unit 2

Requirement Analysis

Requirement

1. The requirement for a system is the description of the services provided by the system and its operational constraints.
2. A requirement is simply a high level, abstract statement of a service that the system should provide or a constraint on the system.

Requirement Engineering

The process of finding out, analyzing, documenting and checking these services and constraints is called requirement engineering.

User Requirements

1. These are statements, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate.
2. The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by system users without detailed technical knowledge.
3. They should only specify the external behavior of the system and should avoid, as for as possible, system design characteristics.
4. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system.

For example

"LIBSYS shall keep track of all data required by copyright licensing agencies in the UK and elsewhere."

However, various problems can arise when requirements are written in natural language sentences in a text document.

a) **Lack of clarity** – It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.
b) **Requirement Confusion** – Functional requirements, non-functional requirements, system goals and design information may be clearly distinguished.
c) **Requirements Amalgamation** – Several different requirements may be expressed together as a single requirement.

As an illustration of some of these problems, consider one of the requirements for the library

"LIBSYS" shall provide a financial accounting system that maintains records of all payments made by users of the system. System managers may configure this system so that regular users may receive discounted rates."

This requirement includes both conceptual and detailed information. It expresses the concept that there should be an accounting system as an inherent part of LIBSYS. However, it also includes the detail that the accounting system should support discounts for regular LIBSYS users. This detail would have been left to the system requirements specification.

To minimize misunderstandings when writing user requirements, you follow some simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. You may also include information on who proposed the requirement (the requirement source) so that you know whom to consult if the requirement has to be changed.
2. Use language consistently. You should always distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using 'shall'. Desirable requirements are not essential and are written using 'should'.
3. Use text highlighting (bold, italic or color) to pick out key parts of the requirement.
4. Avoid, as for as possible, the use of computer jargon. Inevitably, however, detailed technical terms will creep into the user requirements.

System Requirements

1. They set out the system's functions, services and operational constraints in detail.
2. System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design.
3. They add detail and explain how the user requirements should be provided by the system.
4. They may be used as part of the contract for the implementation of the system and should therefore be a complete and consistent specification of the whole system.
5. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

For examples

- On making a request for a document from LIBSYS, the requester shall be presented with a form that records details of the user and the request made.
- LIBSYS request forms shall be stored on the system for five years form the date of the request.
- All LIBSYS request forms must be indexed by user, by the name of the material requested and by the supplier of the request.
- LIBSYS shall maintain a log of all requests that have been made to the system.
- For material where author's lending rights apply, loan details shall be sent monthly to copyright licensing agencies that have registered with LIBSYS.

Functional and Non-Functional Requirements

Software system requirements are often classified as functional, non-functional or domain requirements.

Functional Requirements

1. These are statement of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
2. In some cases, the functional requirements may also explicitly state what the system should not do.
3. These requirements depend on the type of software being developed, the expected users of the software and the general approach taken by the organization when writing requirements.
4. When expressed as user requirements, the requirements are usually described in a fairly abstract way.
5. Functional system requirements describe the system in detail, its inputs and outputs, exceptions, and so on.
6. In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions.

For example, here are examples of functional requirements for a university library system called LIBSYS, used by the students and faculty to order books and documents from other libraries.

a) The user shall be able to search either all of the initial set of databases or select a subset from it.

b) The system shall provide appropriate viewers for the user to read documents in the document store.

c) Every order shall be allocated a unique identifier (ORDER_ID), which the user shall be copy to the document's permanent storage area.

Non Functional Requirements

1. These are constraints on the services or functions offered by the system.
2. They are not directly concerned with the specific functions delivered by the system.
3. They include timing constraints, constraints on the development process and standards.
4. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may define constraints on the system such as the capabilities of I/O devices and the data representations used in system interfaces.
5. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.
6. Failing to meet a non-functional requirement can mean that the whole system is unusable.

For e.g., if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if a real-time control system fails to meet its performance requirements, the control functions will not operate correctly.

7. Non-functional requirements arise through user needs, because of budget constraints, because of organizational policies, because of the need for interoperability with other software or hardware systems, or because of external factors such as safety regulations or privacy legislation.
8. The types of non-functional requirements

a) Product Requirements

- These requirements specify product behavior.
- Examples include performance requirements on how fast the system must execute and how much memory it requires.
- Reliability requirements that set out the acceptable failure rate.
- Portability requirements and usability requirements.

For example

- The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.

b) Organizational Requirements

- These requirements are derived from policies and procedures in the customer's and developer's organization.
- Examples include process standards that must be used.
- Implementation requirements such as the programming language or design method used.
- Delivery requirements that specify when the product and its documentation are to be delivered.

For example

- The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

c) External Requirements

- This broad heading covers all requirements that are derived from factors external to the system and its development process.
- They may include interoperability requirements that define how the system interacts with systems in other organizations.
- Legislative requirements that must be followed to ensure that the system operates within the law.
- Ethical requirements that the system which is developed will be acceptable to its users and the general public.

For example

- The system shall not disclose any personal information about system users apart from their name and library reference number to the library staff who use the system.

Domain Requirements

1. These requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain.
2. They may be functional or non-functional requirements.
3. Domain requirements are derived from the application domain of the system rather than from the specific needs of the system users.
4. They usually include specialized domain terminology or reference to domain concepts.
5. They may be new functional requirements in their own right, constrain existing functional requirements or set out how particular computations must be carried out.
6. Because these requirements are specialized, software engineers often find it difficult to understand how they are related to other system requirements.
7. Domain requirements are important because they often reflect fundamental of the application domain.
8. If these requirements are not satisfied, it may be impossible to make the system work satisfactorily.

For example

The LIBSYS system includes a number of domain requirements:

a) There shall be a standard user interface to all databases that shall be based on the Z39.50 standard.

b) Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manual forwarding to the user or routed to a network printer.

The first requirement is a design constraint. It specifies that the user interface to the database must be implemented according to a specific library standard. The developers therefore have to find out about that standard before starting the interface design. The second requirement has been introduced because of copyright laws that apply to material used in libraries. It specifies that the system must include an automatic delete-on-print facility for some classes in document. This means that users of the library system cannot have their own electronic copy of the document.

Requirement Engineering Process

1. The goal of the requirement engineering process is to create and maintain a system requirement specification (SRS) document.
2. The overall process includes four high-level requirements engineering sub-processes.
3. These are concerned with
   a) Assessing whether the system is useful to the business i.e. **feasibility study**,
   b) Discovering requirements i.e. **requirement elicitation and analysis**,
   c) Converting these requirements into some standard form i.e. **requirements specification,** and
   d) Checking that the requirements actually define the system that the customer wants i.e. **requirement validation.**
4. Fig below illustrates the relationship between these activities. It also shows the documents produced at each stage of the requirements engineering process.

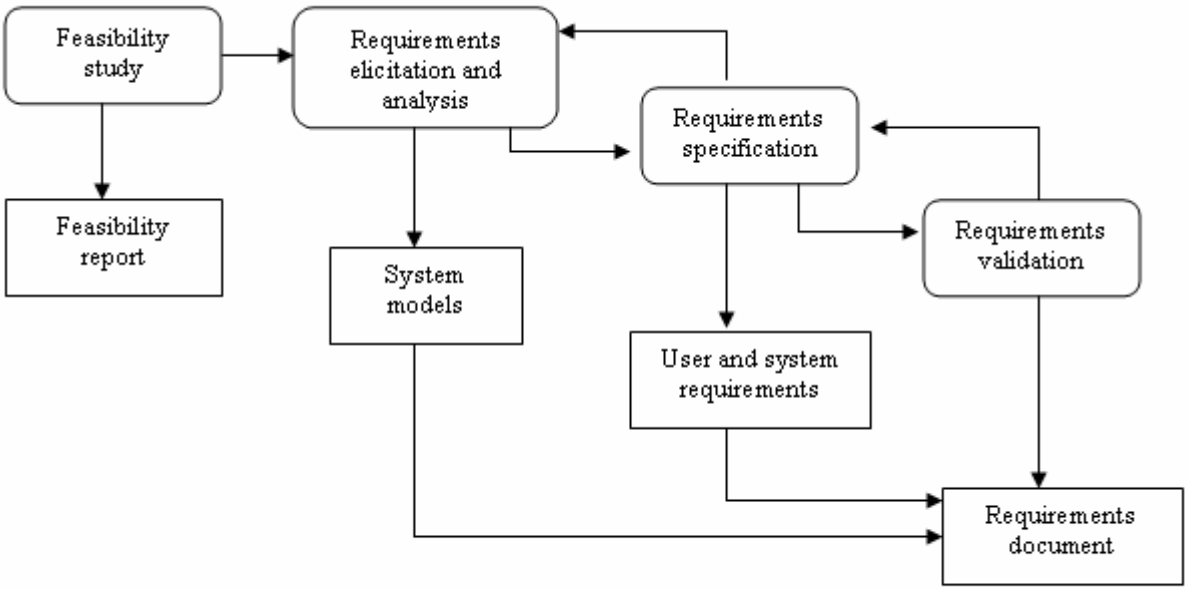


Fig Requirements engineering process

5. The activities shown in above fig are concerned with the discovery, documentation and checking of requirements. In virtually all systems, requirements change. The process involved develop a better understanding of what they want the software to do; the organization buying the system changes; modifications are made to the system's hardware, software and organizational environment. The process of managing these changing requirements is called **requirements management.**

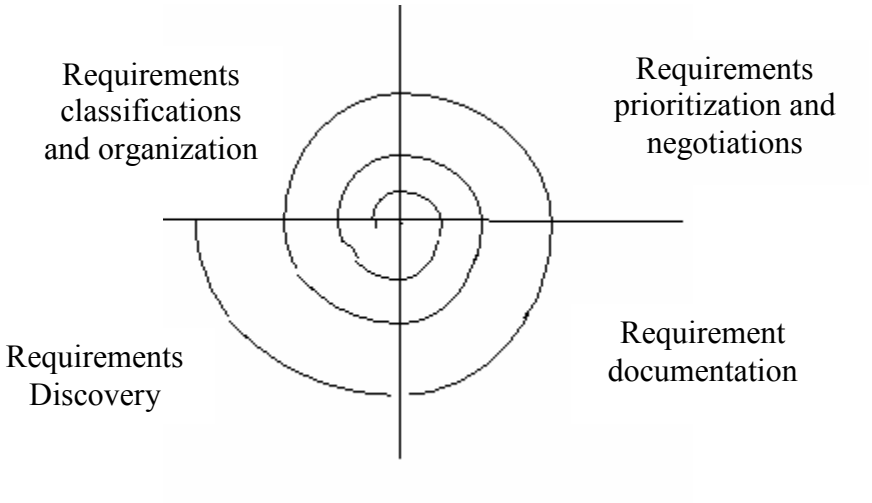
Feasibility Study

1. For all new systems, the requirements engineering process should start with a feasibility study.
2. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes.
3. The results of the feasibility study should be a report that recommends whether or not it is worth carrying on with the requirements engineering and system development process.
4. A feasibility study is a short, focused study that aims to answer a number of questions:
    a) Does the system contribute to the overall objectives of the organization?
    b) Can the system be implemented using current technology and within cost and schedule constraints?
    c) Can the system be integrated with other systems which are already in place?
5. The issue of whether or not the system contributes to business objectives is critical. If a system does not support these objectives, it has no real value to the business.
6. Carrying out a feasibility study involves information assessment, information collection and report writing.
7. The information assessment phase identifies the information that is required to answer these three questions. Once the information has been identified, you should talk with information sources to discover the answers to these questions. Some of the possible questions that may be put are:
    a) How would the organization cope if this system were not implemented?
    b) What are the problems with current processes and how would a new system help to solve these problems?
    c) What direct contribution will the system make to the business objectives and requirements?
    d) Can information be transferred to and from other organizational systems?
    e) Does the system require technology that has not previously been used in the organization?
    f) What must be supported by the system and what need not be supported?

8. In a feasibility study, you may consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of the system that is proposed, technology experts and end-users of the system.
9. Once you have the information, you write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

Requirements elicitation and analysis (Requirement Gathering)

1. In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.
2. Requirements elicitation and analysis may involve a variety of people in an organization. The term stakeholder is used to refer to any person or group who will be affected by the system, directly or indirectly. Stakeholders include end-users who interact with the system and everyone else in an organization that may be affected by its installation. Other system stakeholders may be engineers who are developing or maintaining related systems, business managers, domain experts and trade union representatives.
3. Eliciting and understanding stakeholders requirements is difficult for several reasons:
    a) Stakeholders often don't know what they want from the computer system except in the most general terms. They may find it difficult to articulate what they want the system to do or make unrealistic demands because they are unaware of the cost of their requests.
    b) Stakeholders naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, must understand these requirements.
    c) Different stakeholders have different requirements, which they may express in different ways. Requirements engineers have to consider all potential sources of requirements and discover commonalities and conflict.
    d) Political factors may influence the requirements of the system. For example, managers may demand specific system requirements that will increase their influence in the organization.
    e) The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. Hence the importance may emerge from new stakeholders who were not originally consulted.
4. A very general process model of the elicitation and analysis process is shown in fig below. Each organization will have its own version or instantiation of this general model, depending on local factors such as the expertise of the staff, the type of system being developed and the standards used.
5. The process activities are:
    a) **Requirement discovery** – This is the process of interacting with stakeholders in the system to collect their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
    b) **Requirement classification and organization** – This activity takes the unstructured collection of requirements, group related requirements and organizes them into coherent clusters.
    c) **Requirement prioritization and negotiation** – Inevitably, where multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements, and finding and resolving requirements conflicts through negotiation.
    d) **Requirements documentation** – The requirements are documented and input into the next round of the spiral. Formal and informal requirements documents may be produced.



6. The above fig shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle.

Requirements Validation

1. Requirements validation is concerned with showing that the requirements actually define the system that the customer wants.
2. Requirements validation overlaps analysis in that it is concerned with finding problems with the requirements.
3. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when they are discovered during development or after the system is in service.
4. The cost of fixing a requirements problem by making a system change is much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed and then the system must be tested again.
5. During the requirements validation process, checks should be carried out on the requirements in the requirements document. These check include:
   a) **Validity checks** – A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with distinct needs, and any set of requirements is inevitably a compromise across the stakeholder community.
   b) **Consistency checks** – Requirements in the document should not conflict. That us, there should be no contradictory constraints or descriptions of the same system function.
   c) **Completeness checks** – The requirements document should include requirements, which define all functions, and constraints intended by the system user.
   d) **Realism checks** – Using knowledge of existing technology, the requirements should be checked to ensure that they could actually be implemented. These checks should also take account of the budget and schedule for the system development.
   e) **Verifiability** – To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that are demonstrate that the delivered system meets each specified requirement.
6. A number of requirements validation techniques can be used in conjunction or individually:
   a) **Requirements reviews** – The requirements are concerned systematically by a team of reviewers.
   b) **Prototyping** – In this approach to validation, an executable model of the system is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

c) **Test-case generation** – Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If the test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Requirements Management

1. The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stake holder's understanding of the problem is constantly changing. These requirements must then evolve to reflect this changed problem view.
2. Furthermore, once a system has been installed, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on the organization. Once end-users have experience of a system, they discover new needs and priorities:

a) Large systems usually have a diverse user community where users have different requirements and priorities. These may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

b) The people who pay for a system and the users of a system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery; new features may have to be added for user support if the system is to meet its goals.

c) The business and technical environment of the system changes after installation, and these changes must be reflected in the system. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change with consequent changes in the system support, and new legislation and regulations may be introduced which must be implemented by the system.

3. Requirements management is the process of understanding and controlling changes to system requirements.
4. You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.
5. You need to establish a formal process for making change proposals and linking these to system requirements.
6. The process of requirements management should start as soon as a draft version of the requirements document is available, but you start planning how to manage changing requirements during the requirements elicitation process.

Software Prototyping

1. A prototype is an initial version of the software system which is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions.
2. Rapid development of the prototype is essential so that costs are controlled and users can experiment with the prototype early in the software process.
3. A software prototype supports two requirements engineering process activities:

a) **Requirements Elicitation** – System prototypes allow users to experiment to see how the system supports their work. They get new ideas for requirements and can find areas of strength and weakness in the software. They may then propose new system requirements.

b) **Requirement validation** – The prototype may reveal errors and omissions in the requirements which have been proposed. A function described in a specification may seem useful and well defined. However, when that function is used with others, users often find that their initial view was incorrect or incomplete. The system specification may then be modified to reflect their changed understanding of the requirements.

4. Prototyping can be used as a risk analysis and reduction technique. A significant risk in software development is requirements errors and omissions. The costs of fixing requirements errors at later stages in the process can be very high. Experiments have shown that prototyping reduces the number of problems with the requirement specification. Furthermore, the overall development costs may be lower if a prototype is developed.
5. Prototyping is therefore part of the requirements engineering process.
6. Developing a prototype usually leads to improvements in the specification of the system. Once a prototype is available, it can also be used for other purposes:
   a) User training – A prototype system can be used for training users before the final system has been delivered.
   b) System testing – Prototypes can run 'back-to-back' tests. The same test cases are submitted to the prototype and to the system under test. If both systems give the same result, the test case has not detected a fault. If the results differ, it may mean that there is a system fault and the reasons for the difference should be investigated.

Benefits of Software Prototyping

1. Misunderstanding between software developers and users may be identified as the system functions are demonstrated.
2. Software development staff may find incomplete and/or inconsistent requirements as the prototype is developed.
3. A working, albeit limited, system is available quickly to demonstrate the feasibility and usefulness of the application to management.
4. The prototype may be used as a basis for writing the specification for a production-quality system.
5. Improved system usability.
6. A closer match of the system to the user needs.
7. Improved design quality.
8. Improved maintainability.
9. Reduced development effort.

Prototyping Development Process Model

A process model for prototype development is shown in fig below. The objectives of prototyping should be made explicit from the start of the process. These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements or to develop a system to demonstrate the feasibility of the application to management. The same prototype cannot meet all objectives. If objectives are left implicit, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

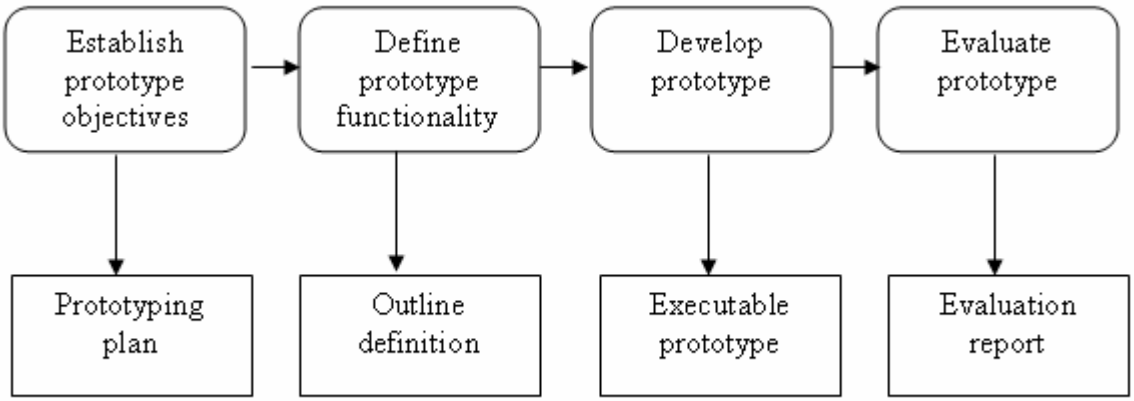


Fig The process of prototype development

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored or may be rudimentary unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation. Users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirement errors and omissions.

Prototyping in the software process

There are two approaches to the prototyping

a) Evolutionary prototyping

1. It is based on the idea of developing an initial implementation, exposing this to user comment and refining this through many stages until an adequate system has been developed.
2. It starts with a relatively simple system which implements the most important user requirements. This is augmented and changed as new requirements are discovered. Ultimately, it becomes the system which is required. There is no detailed system specification and, in many cases, there may not be a formal requirements document.
3. It is now the normal technique used for web-site development and e-commerce applications.
4. The **objective** of evolutionary prototyping is to deliver a working system to end-users. This means that you should normally start with the user requirements which are best understood and which have the highest priority. Lower priority and vaguer requirements are implemented when and if they are demanded by the users.
5. Evolutionary prototyping is part of or has much in common with techniques of Rapid application development (RAD) and Joint Application Development (JAD).

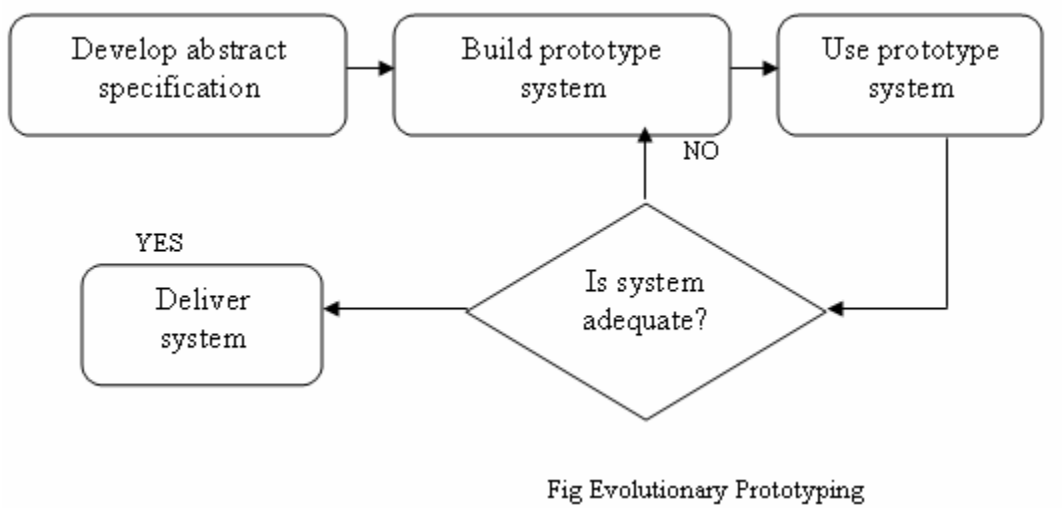


Fig Evolutionary Prototyping

Advantages of Evolutionary Prototyping

1. **Accelerated delivery of the system** – In evolutionary prototyping the delivery of the system is accelerated because the working system is made available in the early stages of the system.
2. **User engaged with the system –** The involvement of users with the development process does not just mean that the system is more likely to meet their requirements. It also means that the end-users of the system have made a commitment to it and are likely to want to make it work.

Problems with Evolutionary Prototyping

1. **Management problems –** Software management structures for large systems are set up to deal with a software process model that generates regular deliverables to assess progress. Prototypes evolve so quickly that it is not cost-effective to produce a great deal of system documentation. Furthermore, rapid prototype development may require unfamiliar technologies to be used. Managers may find it difficult to use existing staff because they lack these skills.
2. **Maintenance problems –** Continual change tends to corrupt the structure of the prototype system. This means that anyone apart from the original developers is likely to find it difficult to understand. Furthermore, if specialized technology is used to support rapid prototype development this may become obsolete. Therefore, finding people who have the required knowledge to maintain the system may be difficult.
3. **Contractual problems –** The normal contractual model between a customer and a software developer is based around a system specification. When there is no such specification, it may be difficult to design a contract for the system development. Customers may be unhappy with a contract which simply pays developers for the time spent on the project as this can lead to function creep and budget overruns; developers are unlikely to accept a fixed-price contract as they cannot control the changes requested by the end-users.

b) Throw-away prototyping

1. This approach extends the requirement analysis process with the intension of reducing overall life-cycle costs.
2. The principle function of the prototype is to clarify requirements and provide additional information for managers to assess process risks.
3. After evaluation, the prototype is thrown away. It is not used for further system development.
4. The **objective** of throw-away prototyping is to validate or derive the system requirements. You should start with those requirements that are not well understood because you need to find out more about them. Requirements that are straightforward may never need to be prototyped.
5. This approach to system prototyping is commonly used for hardware systems.
6. It is not normally used for design validation but to help develop the system requirements. The prototype design is often quite different from that of the final system.
7. The system must be developed as quickly as possible so that users can feed back their prototype experience to the development of the system specification.
8. Functionality may be stripped from the throw-away prototype where these functions are well understood, quality standards may be relaxed and performance criteria ignored.
9. The prototype development language will often be different from the final system implementation language.

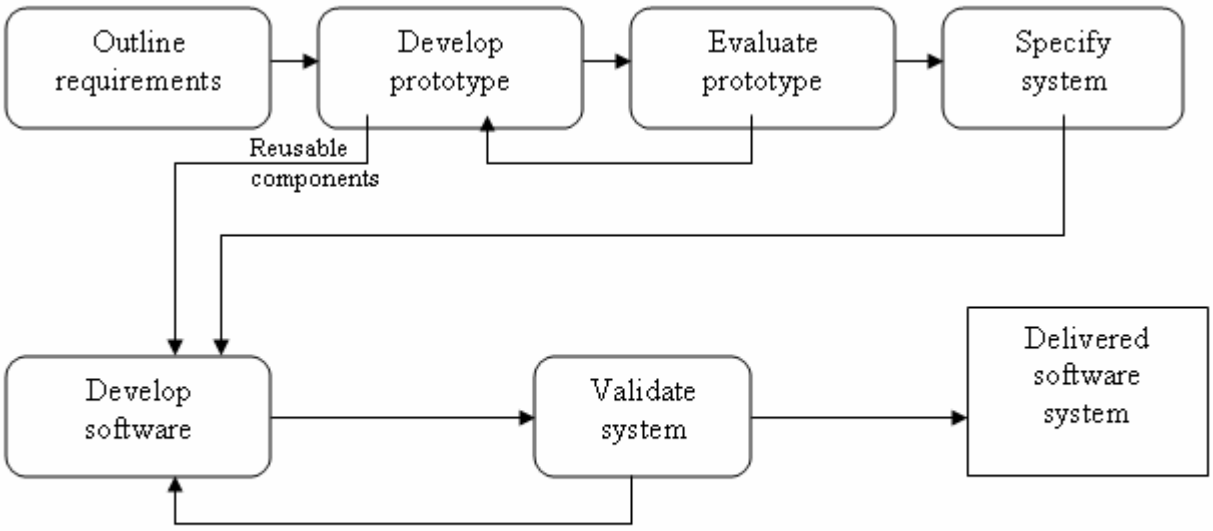


Fig. A software process with throw-away prototype

Difference between Evolutionary Prototype and Throw-away Prototype

No.	Throw-away Prototype	Evolutionary Prototype
1.	The prototype so developed is quick and dirty.	The prototype is not dirty.
2.	This type of prototype is thrown away.	This type of prototype is not thrown away but is evolved further.
3.	Here, we built only difficult parts.	Here, we build the understood parts first on sound foundations.
4.	It involves just one task.	It involves successive steps.
5.	Lesser probability of meeting user's needs.	Higher probability of meeting user needs as we have already used version *i* before using and implementing version $(i+1)^{th}$.

Rapid Prototyping Techniques

Rapid prototyping techniques are development techniques which emphasize speed of delivery rather than other system characteristics such as performance, maintainability or reliability. There are three rapid development techniques that are practical for developing industrial-strength prototypes:

a) Dynamic high-level language development

1. Dynamic high-level languages are programming languages which include powerful run-time data management facilities.
2. These simplify program development because they reduce many problems of storage allocation and management.
3. The language system includes facilities which normally have to be built from more primitive constructs in languages like Ada or C.
4. Examples of very high-level languages are Lisp (based on list structures), Prolog (based on logic) and Smalltalk (based on objects).
5. Until relatively recently, very high-level languages were not widely used for large system development because they need a large run-time support system. This run-time support increases the storage needs and reduces the execution speeds of programs written in the language. However, the increasing power and reducing cost of computer hardware have made these factors less important.

6. Java provides many of the advantages of very high-level languages with the rigor and the opportunities for performance optimization offered by conventional third-generation languages. Many reusable Java components are available so it is clearly a very suitable language for evolutionary prototyping.
7. Table below shows the dynamic languages that are most commonly used for prototype development.

Language	Type	Application domain
Smalltalk	Object-oriented	Interactive Systems
Java	Object-oriented	Interactive Systems
Prolog	Logic	Symbolic processing
Lisp	List-based	Symbolic processing

8. When choosing a prototyping language, you must ask a number of questions:
   - **What is the application domain of the problem?** – As shown in table, different languages are best suited to different application domains. If you want to prototype applications which involve natural language processing, a language such as Lisp or Prolog is more suitable than Java or Smalltalk.
   - **What user interaction is required?** – Different languages provide different levels of support for user interaction. Some languages, such as Smalltalk and Java, are well integrated with web browsers while others, such as Prolog, are best suited to text-based interfaces.
   - **What support environment is provided with the language?** – Mature support environments with many tools and easy access to reusable components simplify the prototype development process.

b) Database programming

1. Evolutionary development is now a standard technique for implementing small and medium-sized applications in the business systems domain. The majority of business applications involve manipulating data from a database and producing outputs which involve organizing and formatting that data.
2. To support the development of these applications, all commercial database management systems now support database programming.
3. Database programming is carried out using a specialized language which embeds knowledge of the database and which includes operations that are geared to database manipulation.
4. The language's supporting environment provides tools to support user interface definition, numeric computation and report generation.
5. The term fourth-generation language (4GL) is used to refer to both the database programming language and its supporting environment.
6. Fourth-generation languages are successful because there is a great deal of commonality across data processing applications. In essence, these applications are concerned with updating a database and producing reports from the information in the database. Standard forms are used for input and output.
7. 4GL are geared towards producing interactive applications which rely on abstracting information from an organizational database, presenting it to end-users on their terminal or workstation and then updating the database with changes made by users.
8. The user interface usually consists of a set of standard forms or a spreadsheet.

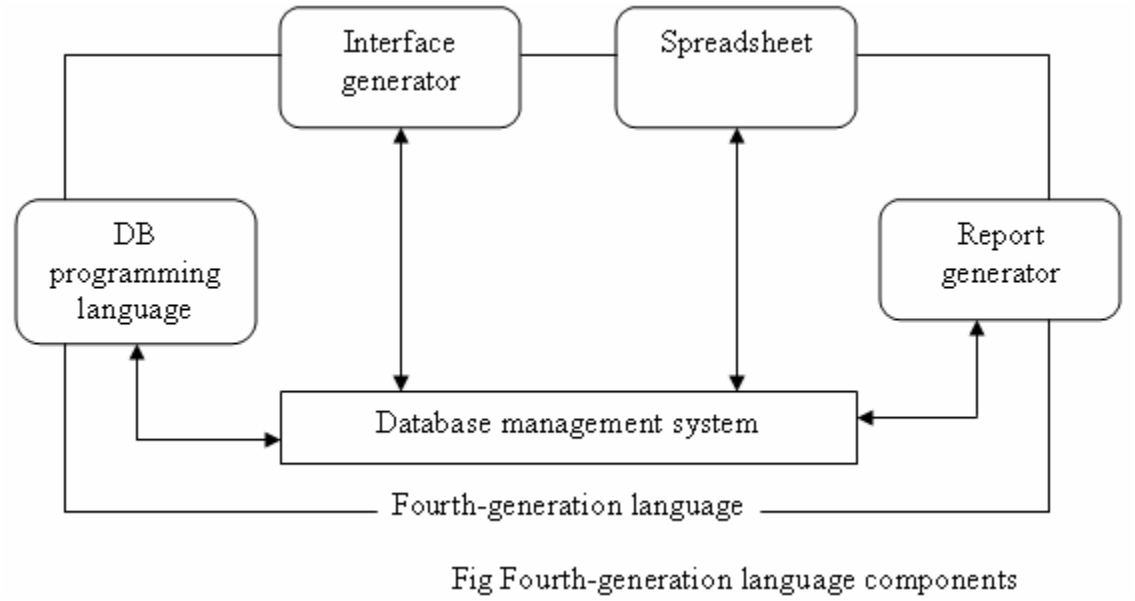


Fig Fourth-generation language components

9. The tools which are included in 4GL environment are:
   - A database query language which is now usually SQL. This may be input directly or generated automatically from forms filled in by an end-user.
   - An interface generator which is used to create forms for data input and display.
   - A spreadsheet for the analysis and manipulation of numeric information.
   - A report generator which is used to define and create reports from information in the database.

10. While 4GLs are very suitable for prototype development, there are some disadvantages in using them for production systems. Programs written in 4GL are usually slower than similar programs in conventional programming languages and usually require much more memory.

c) Component and application assembly

1. The time needed to develop a system can be reduced if many parts of that system can be reused rather than designed and implemented.
2. Prototypes can be constructed quickly if you have a set of reusable components and some mechanism to compose these components into systems.
3. The composing mechanism must include control facilities and a mechanism for component communications.
4. Prototyping with reusable components involves developing a system specification by taking account of what reusable components are available. This may mean that some requirements compromises may have to be made. The functionality of the available components may not be a precise fit for the user requirements. However, user requirements are often fairly flexible so, in most case, this approach can be used for prototype development.

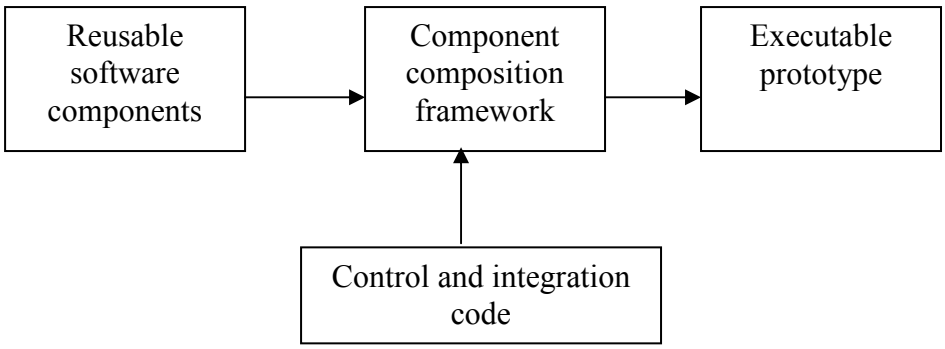


Fig Reusable component composition

5. Prototype development with reuse can be supported at two levels:
   a) The application level where entire application systems are integrated with the prototype so that their functionality can be shared. For example, if the prototype requires a text processing capability, this can be provided by integrating a standard word processing system. Application, such as Microsoft Office applications, support application linking.
   b) The component level where individual components are integrated within a standard framework to implement the system. The standard framework can be a scripting language which is designed for evolutionary development such as Visual Basic, TCL/TK, Python or Perl. Alternatively it can be more general component integration framework based on CORBA, DCOM or JavaBeans.
6. Visual development systems such as Visual Basic support this reuse-based approach to application development. Application programmers build the system interactively by defining the interface in terms of screens, fields, buttons and menus.
7. This approach to system development allows for the rapid development of relatively small and simple applications which can be built by one person or a small team of people. For large systems which must be developed by larger teams it is more difficult to organize.

User Interface Prototyping

1. Graphical user interfaces have now become the norm for interactive systems.
2. The effort involved in specifying, designing and implementing a user interface represents a significant part of application development costs.
3. The designers should not impose their view of an acceptable user interface on users. The users must take part in the interface design process. This realization led to an approach to design called user-centered design that depends on interface prototyping and user involvement throughout the interface design process.
4. From a software engineering point of view, prototyping is an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements. Therefore, evolutionary prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.
5. Interface generators are graphical screen design systems where interface components such as menus, fields, icons and buttons are selected from a menu and positioned on an interface.

Software Requirement Specification (SRS) or Software Requirement Plan (SRP): - It contains the following elements –

1. **Introduction**: - It states the goals and objectives of the software. It represents the scope of the software and the entire project.
2. **Information description**: - It provides a detailed description of the problem that the software has to solve. Here various diagrams are created for representing the relationships, data flow and control flow between various elements (example, DFD).
3. **Functional description**: - It contains the description related to problem solution attained through partitioning and the respective functions to implement the partitioning. It also contains a description of design constraints with justification, performance characteristics and a diagrammatic representation of the overall structure of the software and interrelations among various system elements.
4. **Behavioral description**: - It contains the description related to the internal operation of the software including its response to external events. It may contain various state transition diagrams.
5. **Bibliography**: - The bibliography contains references to various documents that are related to the software, engineering documents, technical references, vendor literature and standards.

Structure of a SRS

The general structure of an SRS proposed by IEEE standard is given below:

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall Description
   - 2.1 Product Perspective
   - 2.2 Product Functions
   - 2.3 User Characteristics
   - 2.4 General Constraints
   - 2.5 Assumptions and Dependencies
3. Specific Requirements
   - 3.1 External Interface Requirements
     - 3.1.1 User Interfaces
     - 3.1.2 Hardware Interfaces
     - 3.1.3 Software Interfaces
     - 3.1.4 Communication Interfaces
   - 3.2 Functional Requirements
     - 3.2.1 Mode 1
       - 3.2.1.1 Functional Requirement 1.1
       - 3.2.1.2 Functional Requirement 1.2
     - 3.2.2 Mode 2
       - 3.2.2.1 Functional Requirement 2.1
       - 3.2.2.2 Functional Requirement 2.2
   - 3.3 Performance Requirements
   - 3.4 Design Constraints
   - 3.5 Attributes
   - 3.6 Other Requirements

The description of various sections of SRS is:

1. **Introduction** – This section contains the purpose, scope, overview, etc. of the requirements document. It also contains the references cited in the document and any definitions that are used.
2. **Overall Description** – It describes the general factors that affect the product and its requirements. Specific requirements are not mentioned, but a general overview is presented to make the understanding of the specific requirements easier. Product perspective is essentially the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principle interfaces of the product are. A general abstract description of the functions to be performed by the product is given.
3. **Specific Requirements** – It describes all the details that the software developer needs to know for designing and developing the system. This is typically the largest and most important part of the document. The external interface requirements section specifies all the interfaces of the software: to people, other software, hardware, and other systems. In the functional

requirements section, the functional capabilities of the system are described. For each functional requirement, the required inputs, desired outputs, and processing requirements will have to be specified.

Benefits of a Great SRS (Need for SRS)

The IEEE 830 standard defines the benefits of a good SRS:

1. ***Establish the basis for agreement between the customers and the suppliers on what the software product is to do.*** The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs. [NOTE: We use it as the basis of our contract with our clients all the time].
2. ***Reduce the development effort.*** The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.
3. ***Provide a basis for estimating costs and schedules.*** The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates. [NOTE: Again, we use the SRS as the basis for our fixed price estimates]
4. ***Provide a baseline for validation and verification.*** Organizations can develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured. [NOTE: We use the SRS to create the Test Plan].
5. ***Facilitate transfer.*** The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.
6. ***Serve as a basis for enhancement.*** Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation. [NOTE: This is often a major pitfall – when the SRS is not continually updated with changes]

Characteristics of a great SRS

An SRS should be

a) **Correct** – An SRS is correct if every requirements included in the SRS represents something required in the final system.

b) **Unambiguous** – An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Requirements are often written in natural language, which are inherently ambiguous. If the requirements are specified in a natural language, the SRS write has to be especially careful to ensure that there are no ambiguities.

c) **Complete** – An SRS is complete if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.

d) **Consistent** – An SRS is consistent if there is no requirement that conflicts with another. The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.

e) **Ranked for Importance** – Generally all the requirements are not of equal importance. Some are critical, others are important but not critical and there are some which are desirable but not very important. Some requirements are core requirements which are not likely to change as time passes, while others are more dependent on time. An SRS should be ranked for importance and / or stability. Stability of a requirement reflects the chances of it changing in future.

f) **Verifiable** – An SRS is verifiable if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement. This implies that the requirement

should have as little subjectivity as possible because subjective requirements are difficult to verify. Don't put in requirements like - "It should provide the user a fast response." Another of my favorites is - "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."

g) **Modifiable** – An SRS is modifiable if its structure and style should be such that any necessary changes can be made easily while preserving completeness and consistency. Presence of redundancy is a major obstacle to modifiability, as it can easily lead to errors. For example, assume that a requirement is stated in two places and that the requirement later needs to be changed. If only one occurrence of the requirements is modified, the resulting SRS will be inconsistent.

h) **Traceable** – An SRS is traceable if the origin of each or its requirements is clear and if it facilitates the referencing of each requirement in future development. Forward traceability means that each requirement should be traceable to some design and code elements. Backward traceability requires that it be possible to trace design and code elements to the requirements they support. Traceability aids verification and validation.

Analysis and Modeling

Models used in the Analysis phase of the development process

The modeling use during analysis phase should achieve three primary objectives –

1) To describe what the customer requires,

2) To establish a basis of software design and

3) To define a set of requirements that can be validated once the software is built.

To attain the above mentioned objectives the following structure for the analysis model is suitable:-

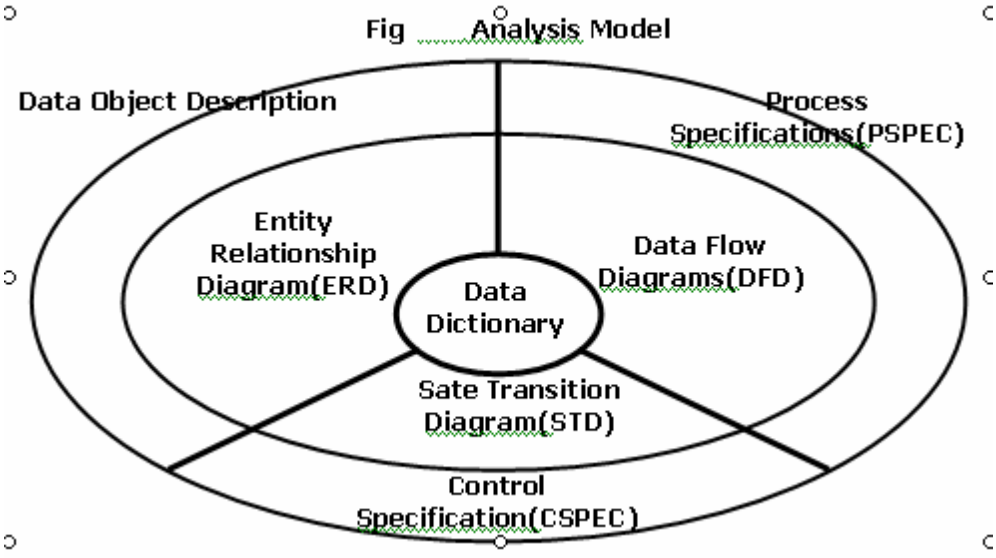


Fig Analysis Model

Entity Relationship Diagram (ERD)

The ERD is used to depict the relationship between the data objects. This notation is used in Data Modeling activity. The attributes of each object can be described using Data Object Description.

Data Modeling: - It identifies the following points related to data objects –

1) The primary data objects to be processed by the system,

2) The composition and attributes of each data object,

3) The current position and visibility of the data objects in the system,

4) The relationship between various data objects and

5) The processes that transform those data objects.

A basic data model consists of three interrelated elements –

1) Data objects,

2) Attributes and

3) Relationships.

1) **Data Objects**: - It's a representation of any composite information that must be understood by the software. Composite information means a collection of different properties or attributes. A data object can be an event (e.g. alarm), a role (e.g. salesperson), an organizational unit (e.g. EDP dept.), a place or a structure (e.g. file) etc. A data object encapsulated data only and there is no reference to the operations that can be performed on it.

2) **Attributes** :- It defines the properties of a data object and can be used for the following purposes – a) Naming an instance of the data object, b) Describe the instance, or c) Make reference to another instance in a different table. The problem context determines the attributes for a data object.

3) **Relationships**: - It's the relevant connection between the any numbers of data objects. While establishing a relationship it must be observed that the relationship is bi-directional.

Understanding Cardinality and Modality during Data modeling

Cardinality: - It's the specification of the number of occurrences of an object that can be related to a number of occurrences of another object. Cardinality is expressed in the terms "one" or "many". Following are the cardinality types:-

a) **One to One (1:1)**:- Here an occurrence of any object 'A' can relate to one and only one occurrence of object 'B' and vice-versa.

b) **One to many (1:N)**:- Here an occurrence of an object 'A' can relate to one or many occurrences of another object 'B', but an occurrence of 'B' can relate to only one occurrence of 'A'.

c) **Many to many (M:N)**:- Here an occurrence of an object 'A' can relate of one or more occurrences of another object 'B' and also an occurrence of 'B' can relate to one or more occurrences of 'A'.

The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar represents "One", and the three-pronged fork indicates "Many".

Modality: - Modality can be either expressed as 1 or 0. The modality is 1 if an occurrence of relationship is mandatory. The modality is 0 if there is no explicit need for the relationship to occur or the relationship is optional. Diagrammatically the symbols used further away from the object rectangles next to the cardinality symbols. Here the vertical bar represents the occurrence of mandatory relationship and a circle represents no explicit need for the relationship.

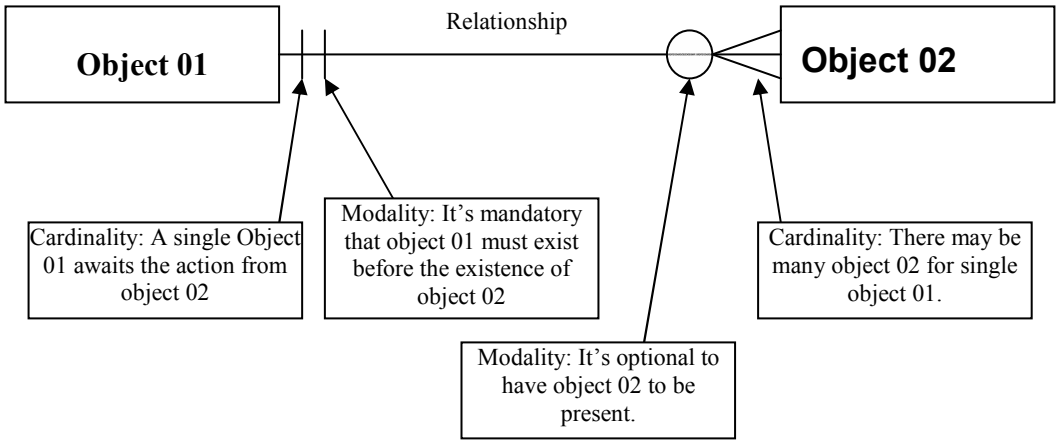


Fig. Entity Relationship Diagram

Guidelines for creating Entity Relationship Diagrams (ERD)

Following are some basic components of an ERD: - a) Data objects, b) Attributes, c) Relationships and d) Various types of indicators. Data objects are represented using rectangles. Relationships are represented using labeled lines, some ERD have a diamond shaped box for representing the relationship. Cardinality and Modality are represented using the above mentioned symbols. The entity relationship diagrams enables a software engineer to fully specify the data objects that go as input into the system and come as out put from a system. It also helps in specifying the properties of the input/output objects and their inter relationships. ERD follow an iterative process of construction and include the following steps:-

1) A list of input and output data objects as well as the entities that consume or produce information is gathered from the customer and also from analyzing what the customer means (knowingly or unknowingly).
2) Taking one object at a time, the customer and the analyst decide whether there exists a connection between the object under scrutiny and other objects.
3) Whenever a connection exists, the analyst and customer create one or more object-relationship pairs.
4) For each object-relationship pair, the respective cardinality and modality are explored.
5) Steps 2 through 4 are continued iteratively until all object-relationship pairs have been defined. During iterations there may be additions and omissions of the object-relationship pairs.
6) After the desired numbers of iterations are completed, the attributes of each entity are defined.
7) The resulting entity-relationship diagram is formalized and reviewed.
8) Steps 1 through 7 are repeated until data modeling is complete.

Data Flow Diagrams (DFD) or Data Flow Graph or Bubble Chart

A DFD is graphical technique that depicts –

a) The information flow and

b) The transformation activities that are applied on the data, as it moves from input to output.

It's very useful during software requirements analysis. The DFDs have basic two levels of development, they are as follows –

1) A Level 0 DFD, also known as Fundamental System Model or Context Model, represents the entire system as a single bubble with incoming arrowed lines as the input data and outgoing arrowed lines as output.

2) A Level 1 DFD might contain 5 or 6 bubbles with interconnecting arrows. Each process represented here is the detailed view of the functions shown in the level 0 DFD.

- Here the rectangular boxes are used to represent the external entities that may act as input or output outside the system.
- Round Circles are used to represent any kind of transformation process inside the system.
- Arrow headed lines are used to represent the data object and its direction of flow.

A Process Specification (PSPEC) can be used to specify the processing details implied by a bubble within a DFD. It also indicates the limitations and restrictions imposed on the process.

Following are some guidelines for developing a DFD:-

1) Level 0 DFD should depict the entire software system as a single bubble.

2) Primary input and output should be carefully noted.

3) For the next level of DFD the candidate processes, data objects and stores should be recognized distinctively.

4) All the arrows and bubbles should be labeled with meaningful names.

5) Information flow continuity should be maintained in all the levels.

6) One bubble should be refined at a time.

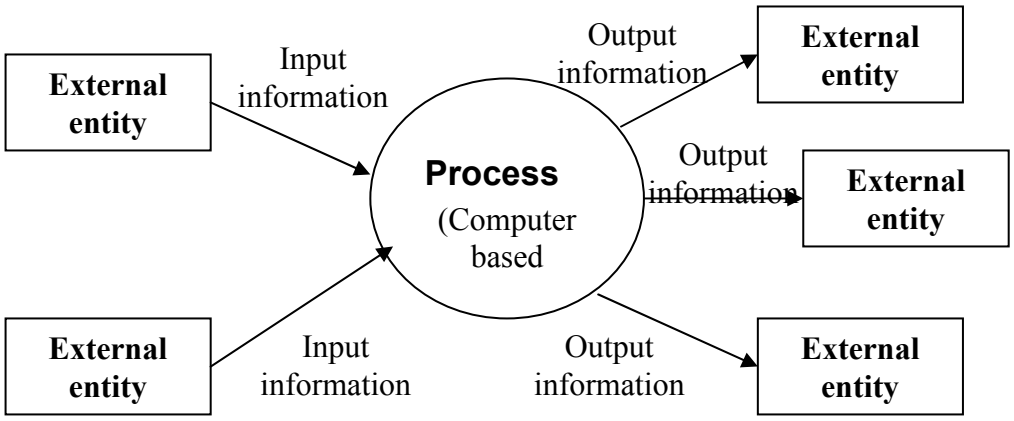


Fig Data Flow Diagram

The Control Specification (CSPEC):- The CSPEC represents the behavior of the system in two different ways –

1) State Transition Diagram (STD), which is a sequential specification of the behavior of the system in terms of state transitions between various processes in the system as bubbles and arrowed lines,

2) Process Activation Table (PAT), which is a table containing the explanation of the states in the STD, in terms of processes.

State Transition Diagrams

Transition tables: - They are used to specify changes in the state of a system as a function of the driving forces. The state of a system summarizes the status of all the entries in the system at a particular time. Given the current state and current conditions the next state is evaluated.

A simple transition table can be evaluated as f(Si, Cj) = Sk, where Si – is the present state, Cj – is the present condition , Sk – is the resultant state acquired by the system due to the present Si & Cj.

Current State	Current Input	
	A	b
State 0	Attained State 0	Attained State 1
State 1	Attained State 1	Attained State 0
	Next State	

From the above table we can derive a conclusion that when the system is in the present state, for ex. 'State 0' and the input given to the system is 'a' then the system would attain a state 'Attained State 0' and remain in it until there are some changes in the input.

1) **Finite State Mechanisms**: - Data flow diagrams, regular expressions and transition tables can be combined to provide a powerful finite state mechanism for functional specification of the software systems.

For example we assume the following transition table and create a transition diagram

Current State	Current Input	
	A	b
S0	S0	S1
S1	S1	S0
	Next State	

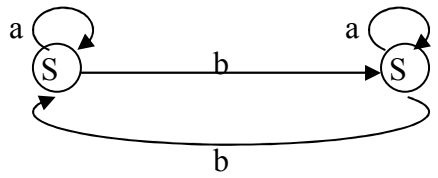


Fig State Transition Diagram

Data Dictionary: - The data dictionary is an organized collection of all data elements related to a system. It contains precise and complete definitions such that the user and the system analyst will have a common understanding of inputs, outputs, storage components and intermediate calculations. Some common information found in data dictionaries are as follows:-

1) **Name**: - It's the information by which an element in the system can be distinctively recognized.

2) **Alias**: - It's a duplicate name for any element.

3) **Where-used / how-used**: - It's the listing of the processes that use a particular element and the manner or mode in which the element is used.

4) **Content description**: - It contains some standard notations for representing the content of any element.

5) **Supplementary notations**: - It contains other information about data types, present value, restrictions, and limitations etc, related to elements of the system.

Advantages of Data Dictionary

1. **It is a mechanism for name management** – Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.

2. **It serves as a store of organizational information** – As the system is developed information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

Difference between DFD and ERD

No	DFD	ERD
1	It is a graphical technique that represents information flow and the transforms that applied as data move form input to output.	It is a graphical technique that represents data objects their relations and associations.
2	It provides a mechanism for functional modeling as well as information flow modeling.	It provides a mechanism for data modeling.
3	It does not contain the relationships among data.	It contains the relationships among various entities.
4	Inputs and outputs are represented separately.	Inputs and outputs are not represented separately.
5	Classes/objects cannot be clear from DFD i.e. Which data table should be kept together.	Classes/objects are clear from ERD i.e. which data table should be kept together.
6	It can give detailed functional overview of a system.	It cannot give detailed functional overview of a system.

Difference between DRD and Flow Chart

No	DFD	Flow Chart
1.	It is a graphical representation of data flow in the system.	It is a graphical representation for procedural design.
2.	It depicts the functions and sub-functions that transform data flow.	It cannot depict the sub-functions.
3.	It cannot depict conditions and loops.	It depicts conditions and loops.
4.	It provides a mechanism for functional modeling as well as information modeling.	It provides a mechanism for functional modeling.
5.	It may be used to represent a system or software at any level of abstraction.	It may be used to represent a system at only one level of abstraction.

Unit 3

Design concepts and principles

Software Design

1. Software design deals with transforming the customer requirements, as described by the SRS document, into a form that is implementable using a programming language.
2. For a design to be easily implementable in a conventional programming language, the following items must be designed during the design phase.
    - Different modules required to implement the design solution.
    - Control relationships among the identified modules. The relationship is also known as the call relationship or invocation relationship among modules.
    - Interface among different modules. The interface among different modules identifies the exact data items exchanged among the modules.
    - Data structures of the individual modules.
    - Algorithms required to implement the individual modules.
3. Thus, the objective of the design phase is to take the SRS document as the input and produce the above mentioned documents before completion of the design phase.
4. We can broadly classified the design activities into two important parts:
    a) **Preliminary (or High level) Design**
        - High level design means identification of different modules, the control relationships and the definitions of the interfaces among them.
        - The outcome of the high level design is called the ***program structure*** or ***software architecture***.
        - Many different types of notations have been used to represent a high level design. A popular way is to use a tree like diagram called the ***structure chart*** to represent the control hierarchy in a high level design.
    b) **Detailed Design**
        - During detailed design, the data structure and the algorithms of different modules are designed.
        - The outcome of the detailed design stage is usually known as the ***module specification*** *document*.

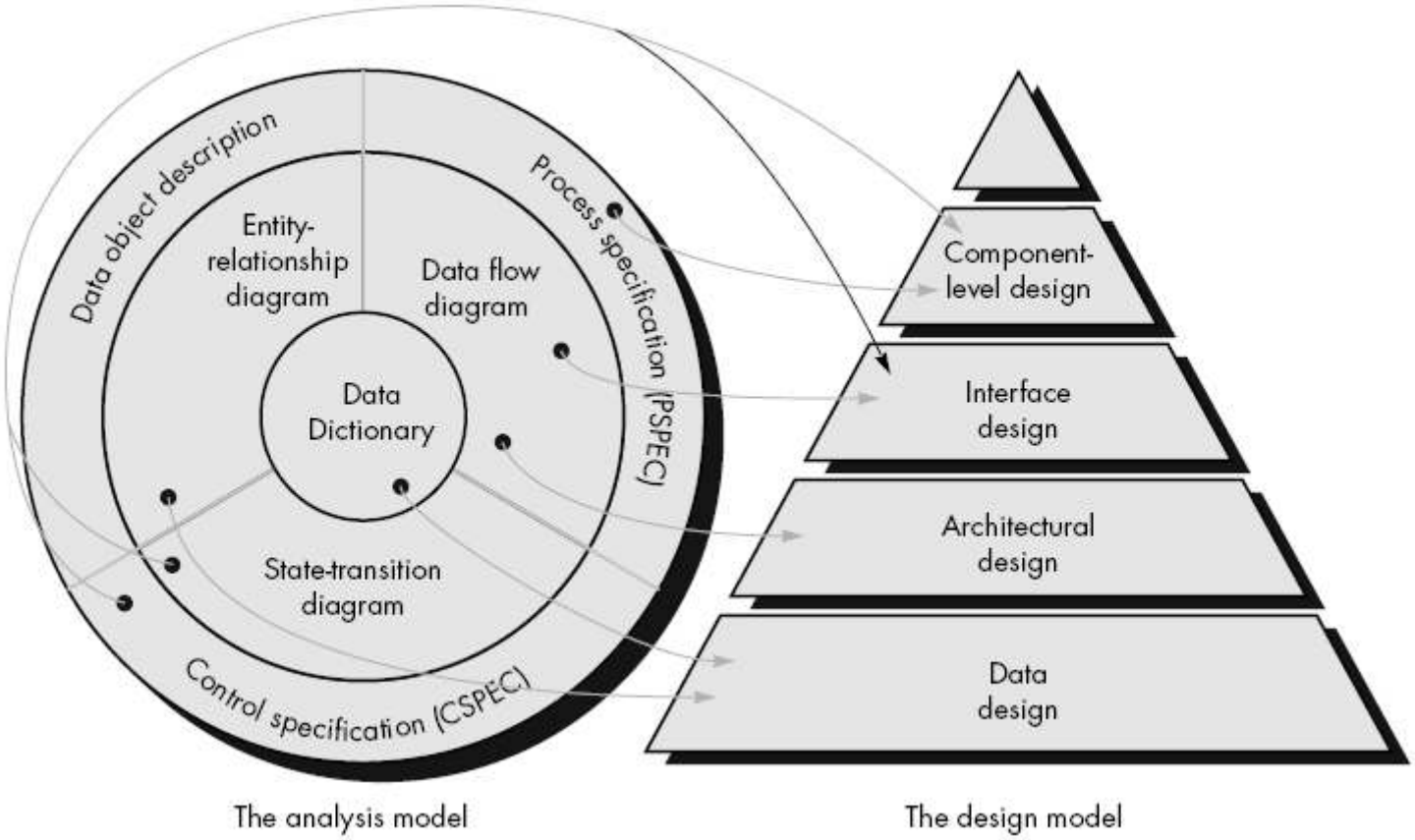
Difference between Good Design and a Bad Design

No.	Good Design	Bad Design
1.	If the design is good then it will not exhibit ripple effect i.e. change in one part of system will not affect other parts of the system.	A bad design will show ripple effect.
2.	It will be simple.	It will be complex.
3.	System can be extended with changes in one place.	It can't add a new function without only breaking an existing function.
4.	The logic is near the data it operates on.	We can't remember where all the implicitly linked changes have to take place.
5.	A good design costs less.	A bad design has more cost.
6.	No need of logic duplication.	Logic has to be duplicated.

Design Model

The design principles and concepts establish a foundation for the creation of the design model that encompasses representation of data, architecture, interface and components. Like the analysis model before it, each of these design representations is tied to the others, and all can be traced back to software requirements.

The Entity-Relationship Diagrams (ERD), the Data Flow Diagrams (DFD), the State Transition Diagrams (STD) and the Data Dictionaries (DD) that are constructed during the requirements phase are directly mapped on to the corresponding design model as shown below.



1. **Data Design** – It transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects (or entities) and the relationships defined in ER diagram and the detailed data content depicted in the Data Dictionary provide the basis for the data design activity. Detailed data design occurs as each software component is designed.
2. **Architectural Design** – It defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system. This design representation forms the framework of a computer based system. It can be derived from the system specification, the analysis model and the interaction of subsystems defined within the analysis model.
3. **Interface Design** – It describes how the software communicates within itself, with systems that interoperate with it and with humans who use it. An interface implies a flow of information and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.
4. **Component Level Design** – It transforms structural elements of the software architecture into a procedural description of software components. Information obtained from ER-Diagrams, Data Flow diagrams or STDs, serves as the basis for component design.

During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word – ***Quality***. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be accessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, we risk building an unstable system – one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be accessed until late in the software process, when time is short and many dollars have already been spent.

Design Process

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software i.e., the design is represented at a high level of abstraction – a level that can be directly traced to the specific system objectives. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

McGlaughlin suggests 3 characteristics that serve as a guide for evaluation of a good design:

1. The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
2. The design must be readable, understandable guide for those who generate code and for those who test and support the software.
3. The design should provide a complete picture of the software, addressing all data, functional and behavioral domains.

Each of these characteristics is actually a goal of the design process.

Quality Guidelines that lead to a good design

1. A design should exhibit an architecture that
   a) Has been created using recognizable architectural styles or patterns;
   b) Is composed of components that exhibit good design characteristics, and
   c) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to the components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Design Principles

Software design is both a process and a model. The 'design process' is a sequence of steps that enable the designer to describe all aspects of the software to be built. The 'design model' is however, an equivalent of an architect's plan for a house. It begins by representing the totality of the thing to be built (e.g. a 3D house) and slowly refining it into more details. Similarly, the design model that is created for software provides a variety of different views of the computer software.

Davis – Design Principles

1. The design process should not suffer from "tunnel vision". A good designer should consider alternative approaches, judging each based on the requirements of the problem, and the resources available to do the job.
2. The design should be traceable to the analysis model. It is necessary to have a means for tracking how requirements have been satisfied by the design model.
3. The designer should not reinvent the wheel, i.e., Use the set of design patterns, already encountered so that new patterns are not reinvented. Time is short and resources are limited. Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world i.e., the structure of the software design should mimic the structure of the problem domain.
5. The design should exhibit uniformity and integration. A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
6. The design should be structured to accommodate change.
7. The design should be structured to degrade gently, even when aberrant data, events or operating conditions are encountered. Well designed software should never "bomb". It should be designed to accommodate unusual circumstances and if it must terminate processing, do so in a graceful manner.
8. Design is not coding, coding is not design.
9. The design should be assessed for quality as it is being created, not after the fact.
10. The design should be reviewed to minimize conceptual (semantic) errors.

Design Concepts

There are 9 design concepts that we must study:

1. Abstraction
2. Refinement
3. Modularity
4. Software Architecture
5. Control Hierarchy
6. Structural Partitioning
7. Data Structure
8. Software Procedure
9. Information Hiding

1. Abstraction

a) When we consider a modular solution to any problem, many ***levels of abstraction*** can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation – oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

b) Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

c) As we move through different levels of abstraction, we work to create procedural and data abstractions. A ***procedural abstraction*** is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word ***open*** for a door. ***Open*** implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

d) A ***data abstraction*** is a named collection of data that describes a data object. In the context of the procedural abstraction ***open***, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction ***open*** would make use of information contained in the attributes of the data abstraction **door**.

e) Many modern programming languages provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.

f) ***Control abstraction*** is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the ***synchronization semaphore*** used to coordinate activities in an operating system.

2. Refinement

a) ***Stepwise refinement*** is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

b) Refinement is actually a process of ***elaboration.*** We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

c) Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

3. Modularity

a) It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

b) Let C(x) be a function that defines the perceived complexity of a problem x, and
E(x) be a function that defines the effort (in time) required to solve a problem x.
For two problems, p1 and p2, if $C(p1) > C(p2)$
it follows that $E(p1) > E(p2)$
i.e., it does take more time to solve a difficult problem.
Also, from experimentation it has been found that $C(p1 + p2) > C(p1) + C(p2)$
i.e., the perceived complexity of a problem that combines p1 and p2 is greater than the perceived complexity when each problem is considered separately. So,
$E(p1 + p2) > E(p1) + E(p2)$
This leads to a "divide and conquer" conclusion—i.e., it is easier to solve a complex problem when you break it into manageable pieces.

i.e., if we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid.

c) Consider the following graph

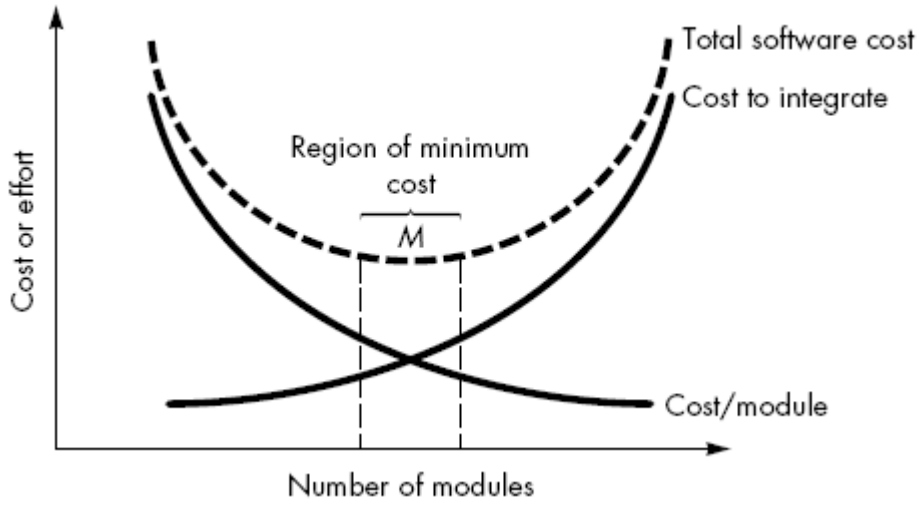


Fig Total cost for Efforts curves

i.e., the effort (cost) required to develop an individual software module does decrease as the total number of modules increases, (cost/module decreases as the number of modules increases). Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the module also grows. These characteristics lead to a total cost or effort curve as shown in fig above. There is a number, M, of modules that would result in minimum development cost but we do not have the necessary sophistication to predict M with assurance. The curves above do provide a useful guidance when modularity is considered. We should modularize but care should be taken to stay in the vicinity of M. Under-modularity or over-modularity should be avoided.

4. Software Architecture

a) It covers the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. So, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components.

b) An architectural design can be represented using any of the 5 – models given below:

- ***Structural Models*** – They represent architecture as an organized collection of program components.
- ***Framework Models*** – They increase the level of design abstraction by attempting to identify repeatable architectural design frameworks.
- ***Dynamic Models*** – They address the behavioral aspects of the program architecture (states).
- ***Process Models*** – They focus on the design of the business or technical process that the system must accommodate.
- ***Functional Models*** – They can be used to represent the functional hierarchy of a system.

c) A number of different architectural description languages (ADLs) have been developed to represent these models.

5. Control Hierarchy

a) It is also called as ***program structure.***

b) It represents the organization of program components (modules) and implies a ***hierarchy of control.***

c) It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions or repetitions of operations nor is it necessarily applicable to all architectural styles.

d) The most commonly used notation to represent control hierarchy is the tree–like diagram that represents hierarchical control for call and return architectures.

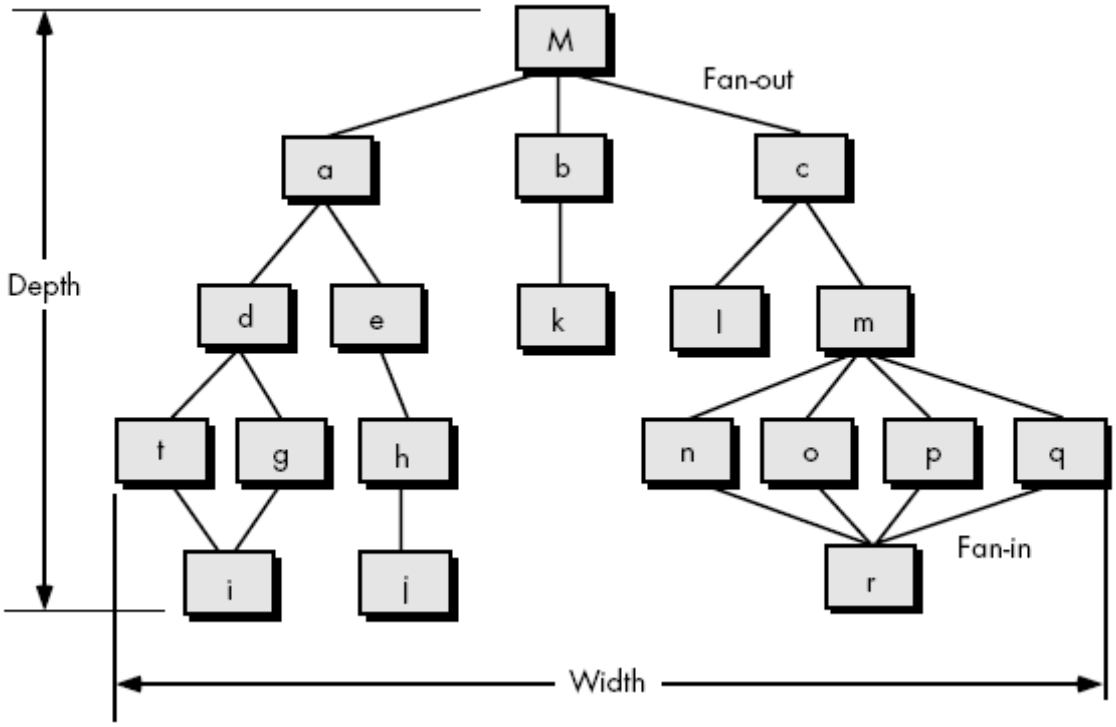


Fig Structural terminology for a call and return architectural style

e) ***'Depth'*** and ***'Width'*** provide an indication of the number of levels of control and overall span control respectively.

f) ***'Fan-out'*** is a measure of the number of modules that are directly controlled by another module. For e.g. Fan-out of M is 3.

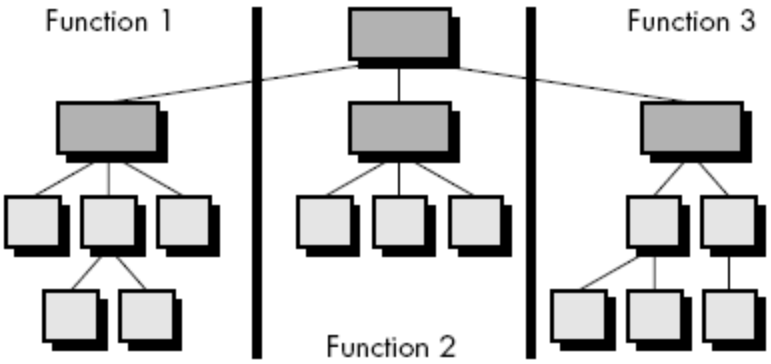
g) ***'Fan-in'*** indicates how many modules directly control a given module. For e.g. Fan-in of r is 4.

h) A module that controls another module is said to be ***super ordinate*** to it and conversely, a module controlled by another is said to be ***subordinate*** to the controller. For e.g. module M is super ordinate to modules a, b and c. Module h is subordinate to module e and is ultimately subordinate to module M.

6. <u>Structural Partitioning</u>

If the architectural style of a system is hierarchical, then the program structure can be partitioned both – horizontally and vertically.

a) **Horizontal Partitioning** – It defines separate branches of the modular hierarchy for each major program function.



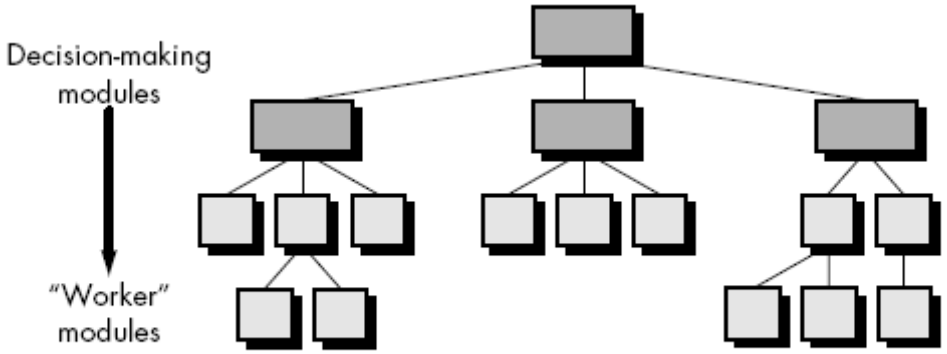
(a) Horizontal partitioning

Control modules, represented in darker – shade (hatched ones) are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines 3 partitions input, data transformation (or processing) and output. It has many benefits:

i. Software that is easier to test.
ii. Software that is easier to maintain.
iii. Propagation of fewer side effects.
iv. Software that is easier to extend.

Its negative side (drawback) is that it causes data to be passed across module interfaces and can complicate the overall control of program flow.

b) **Vertical Partitioning** – Also called as ***factoring***, suggests that control and work should be distributed top-down in the program structure. Top – level modules should perform control functions and do little actual processing work. Whereas the modules that reside low in the structure should be the workers, performing all input, computation and output tasks.



(b) Vertical partitioning

So, it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to the modules that are subordinate to it, Whereas a change to a worker module (at low level) is less likely to cause the propagation of side effects. In general, changes to computer programs resolve a round changes to input, computation (or transformation) and output. The overall control structure of the program is far less likely to change. For this reason, vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable – a key quality factor.

7. <u>Data Structure</u>

a) Data structure is a representation of the logical relationship among individual elements of data.
b) Data structure dictates the organization, methods of access, degree of associativity and processing alternatives for information.
c) It may be a scalar item (or a variable), a sequential vector (array) or a linked list.
d) Note that data structures like program structures can be represented at different levels of abstraction.

8. <u>Software Procedure</u>

a) It focuses on the processing details of each module individually.
b) Procedures must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations and even data organization and structure.
c) A procedural representation of software is layered i.e., we will have procedure for super ordinate modules first and then for subordinate modules.

9. Information Hiding

a) It suggests that the modules should be specified and designed so that information (procedure and data) contained within a module is not accessible to other modules that have no need for such information.

b) Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

c) Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

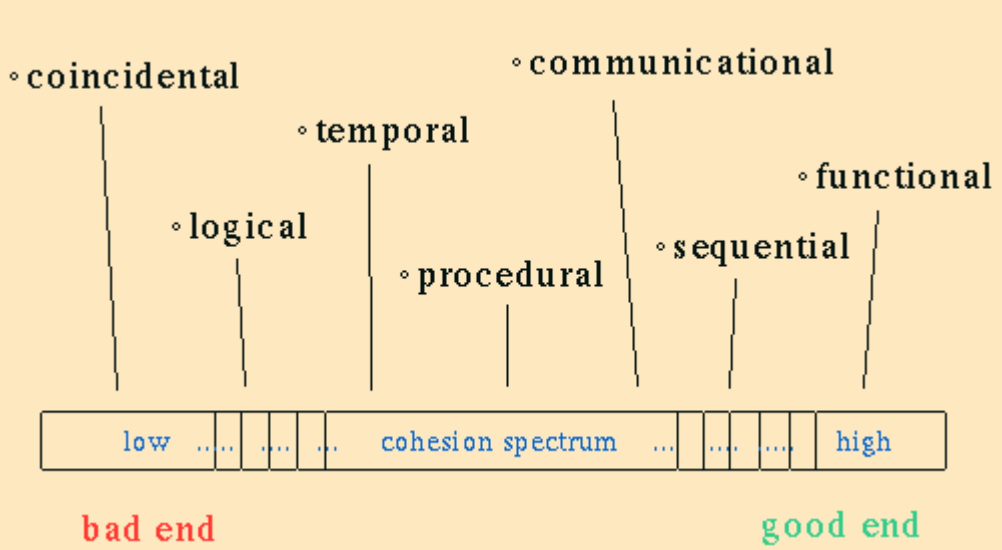
Design documentation

1. The ***Design Specification*** addresses different aspects of the design model and is completed as the designer refines his representation of the software. First, the overall scope of the design effort is described. Much of the information presented here is derived from the ***System Specification*** and the analysis model ***(Software Requirements Specification)***.
2. Next, the data design is specified. Database structure, any external file structures, internal data structures, and a cross reference that connects data objects to specific files are all defined.
3. The architectural design indicates how the program architecture has been derived from the analysis model. In addition, structure charts are used to represent the module hierarchy (if applicable).
4. The design of external and internal program interfaces is represented and a detailed design of the human/machine interface is described. In some cases, a detailed prototype of a GUI may be represented.
5. Components—separately addressable elements of software such as subroutines, functions, or procedures—are initially described with an English-language processing narrative. The processing narrative explains the procedural function of a component (module). Later, a procedural design tool is used to translate the narrative into a structured description.
6. The ***Design Specification*** contains a requirements cross reference. The purpose of this cross reference (usually represented as a simple matrix) is (a) to establish that all requirements are satisfied by the software design and (b) to indicate which components are critical to the implementation of specific requirements.
7. The first stage in the development of test documentation is also contained in the design document. Once program structure and interfaces have been established, we can develop guidelines for testing of individual modules and integration of the entire package. In some cases, a detailed specification of test procedures occurs in parallel with design. In such cases, this section may be deleted from the ***Design Specification***.
8. Design constraints, such as physical memory limitations or the necessity for a specialized external interface, may dictate special requirements for assembling or packaging of software. Special considerations caused by the necessity for program overlay, virtual memory management, high-speed processing, or other factors may cause modification in design derived from information flow or structure. In addition, this section describes the approach that will be used to transfer software to a customer site.
9. The final section of the ***Design Specification*** contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents, and other relevant information are presented as a special note or as a separate appendix. It may be advisable to develop a ***Preliminary Operations/Installation Manual*** and include it as an appendix to the design document.

Modular Design

Cohesion

Cohesion is the measure of strength of the association of elements within a module. Modules whose elements are strongly and genuinely related to each other are desired. A module should be highly cohesive.



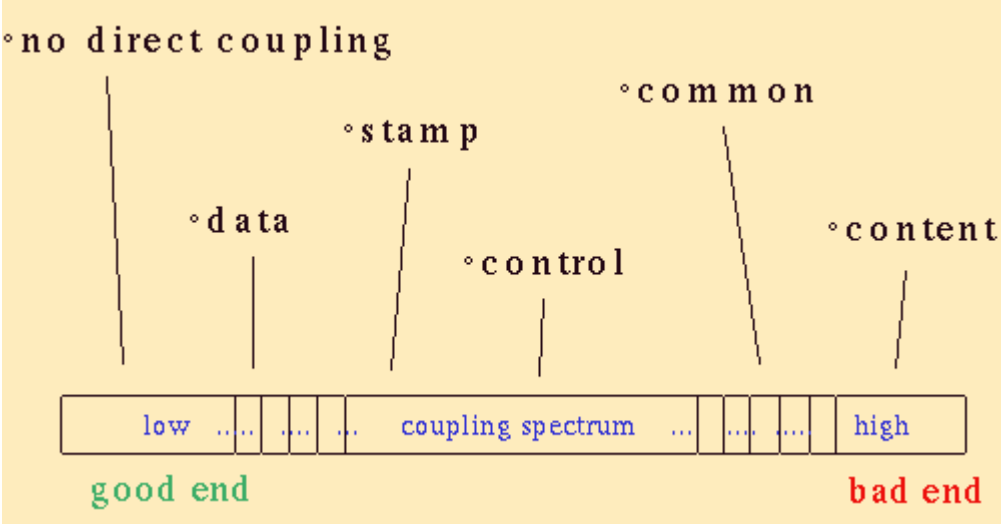
Types of Cohesion

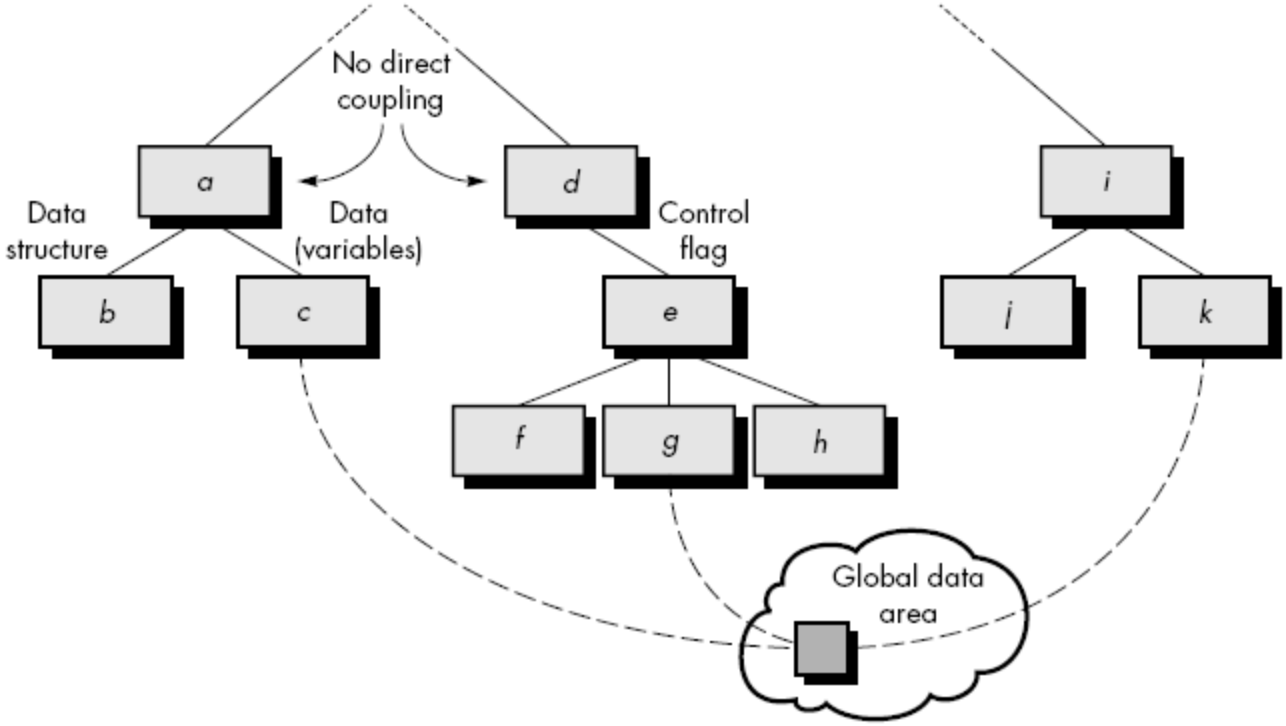
There are 7 types of cohesion in a module

1. **Coincidental Cohesion** – A module has coincidental cohesion if its elements have no meaningful relationship to one another. It happens when a module is created by grouping unrelated instructions that appear repeatedly in other modules.
2. **Logical Cohesion** – A logically cohesive module is one whose elements perform similar activities and in which the activities to be executed are chosen from outside the module. Here the control parameters are passed between those functions. For example, Instructions grouped together due to certain activities, like a switch statement. For ex. A module that performs all input & output operations.
3. **Temporal Cohesion** – A temporally cohesive module is one whose elements are functions that are related in time. It occurs when all the elements are interrelated to each other in such a way that they are executed a single time. For ex. A module performing program initialization.
4. **Procedural Cohesion** – A procedurally cohesive module is one whose elements are involved in different activities, but the activities are sequential. Procedural cohesion exists when processing elements of a module are related and must be executed in a specified order. For example, Do-while loops.
5. **Communication Cohesion** – A communicationally cohesive module is one whose elements perform different functions, but each function references the same input information or output. For example, Error handling modules.
6. **Sequential Cohesion** – A sequentially cohesive module is one whose functions are related such that output data from one function serves as input data to the next function. For example, deleting a file and updating the master record or function calling another function.
7. **Functional Cohesion** – A functionally cohesive module is one in which all of the elements contribute to a single, well-defined task. Object-oriented languages tend to support this level of cohesion better than earlier languages do. For example, When a module consists of several other modules.

Coupling

Coupling is the measure of the interdependence of one module to another. Modules should have low coupling. Low coupling minimizes the "ripple effect" where changes in one module cause errors in other modules.





Types of Coupling

There are 6 types of coupling in the modules are

1. **No direct Coupling** – These are independent modules and so are not really components of a single system. For e.g., this occurs between modules a and d.
2. **Data Coupling** – Two modules are data coupled if they communicate by passing parameters. This has been told to you as a "good design principle" since day one of your programming instruction. For e.g., this occurs between module a and c.
3. **Stamp Coupling** – Two modules are stamp coupled if they communicate via a passed data structure that contains more information than necessary for them to perform their functions. For e.g., this occurs between modules b and a.

4. **Control Coupling** – Two modules are control coupled if they communicate using at least one "control flag". For e.g., this occurs between modules d and e.
5. **Common Coupling** – Two modules are common coupled if they both share the same global data area. Another design principle you have been taught since day one: don't use global data. For e.g., this occurs between modules c, g and k.
6. **Content Coupling** – Two modules are content coupled if:
   i. One module changes a statement in another (Lisp was famous for this ability).
   ii. One module references or alters data contained inside another module.
   iii. One module branches into another module.

   For e.g., this occurs between modules b and f.

Difference between Cohesion and Coupling

No	Cohesion	Coupling
1.	Cohesion is the measure of strength of the association of elements within a module.	Coupling is the measure of the interdependence of one module to another.
2.	A module should be highly cohesive.	Modules should have low coupling.
3.	Cohesion is a property or characteristics of an individual module.	Coupling is a property of a collection of modules.
4.	The advantage of cohesion is the ability to avoid changing source and target systems just to facilitate integration.	The advantage of coupling is the ability to bind systems by sharing behavior, and bound data, versus simple sharing information.
5.	The fact that a single system failure won't bring down all connected systems.	The fact that systems coupled could cease to function if one or more of the coupled systems go down.

Design Heuristics

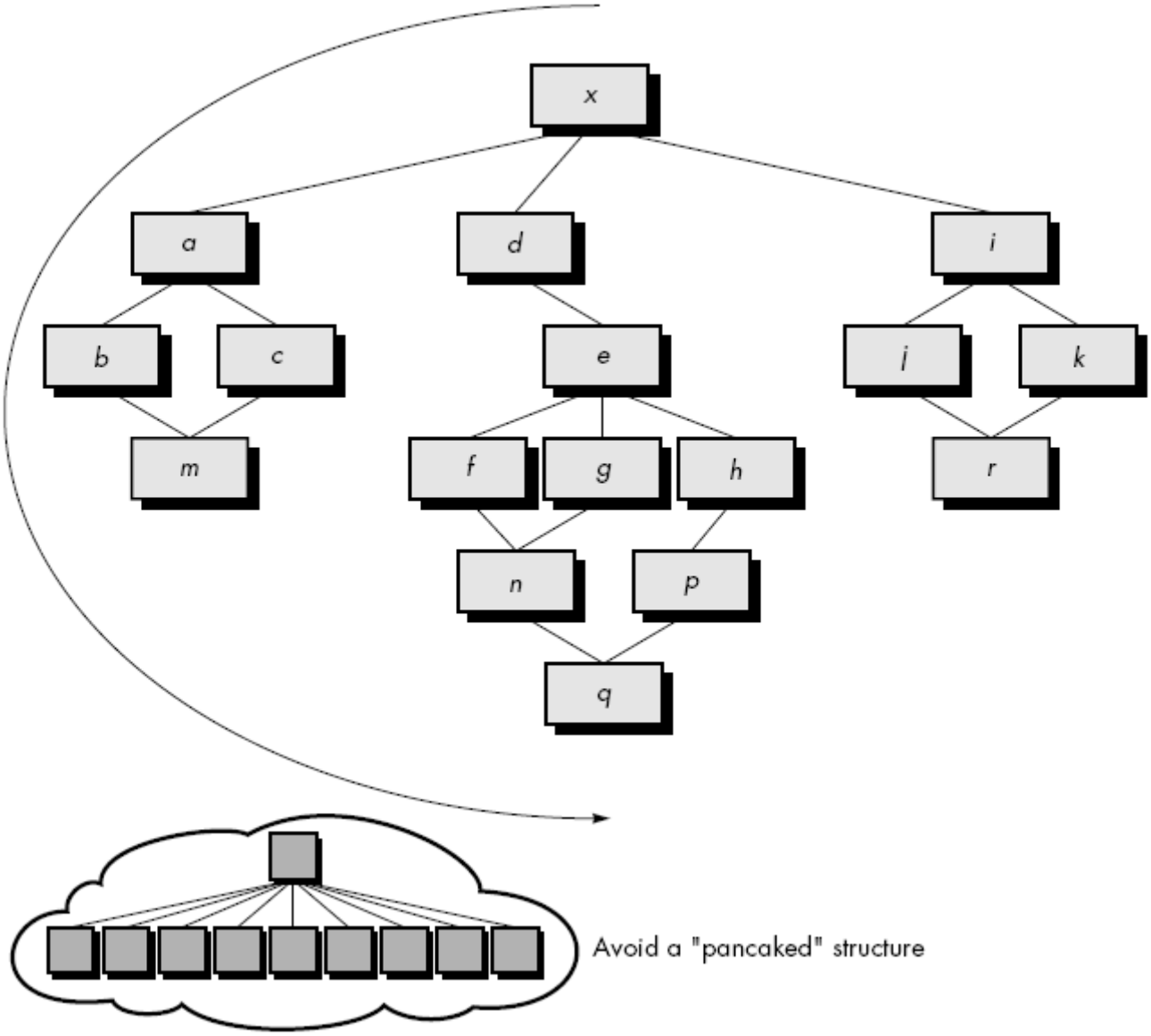
Once program structure has been developed, effective modularity can be achieved by applying the design concepts. The program structure can be manipulated according to the following set of heuristics:

1. Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.

Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure. An imploded module is the result of combining the processing implied by two or more modules. An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.

The structure shown inside the cloud in Figure below does not make effective use of factoring. All modules are "pancaked" below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.



3. Keep the scope of effect of a module within the scope of control of that module.

The scope of effect of module e is defined as all other modules that are affected by a decision made in module e. The scope of control of module e is all modules that are subordinate and ultimately subordinate to module e. Referring to Figure above, if module e makes a decision that affects module r, we have a violation of this heuristic, because module r lies outside the scope of control of module e.

4. Evaluate module interfaces to reduce complexity and redundancy and improve consistency.

Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be reevaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive.

A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use. A module that restricts processing to a single sub-function exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. **Strive for "controlled entry" modules by avoiding "pathological connections."**

This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

Architectural Design

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.

So, architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

1. Analyze the effectiveness of the design in meeting its stated requirements.
2. Consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. Reducing the risks associated with the construction of the software.

Why is architecture important? 3 reasons are:

1. Representations of software architecture are an enabler for communication between all parties interested in the development of a computer – based system.
2. Architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
3. Architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together.

Principles of Data design – by Wasserman

Wasserman proposed the following principles:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low – level data design decisions should be deferred until late in design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

Design Notations

In software design the representation schemes are of fundamental importance. A good design can clarify the interrelationships and actions of interest, while poor notation can compliance and interfere with good practice. At least three levels of design specifications exist: external design specifications, which describe the external characteristics of a software system; architectural design specifications, which describe the structure of the system; and detailed design specifications, which describe control flow, data representation, and other algorithmic details within the modules. Some common design notations are as follows:

Data flow diagrams (Bubble chart)

1) These are directed graphs in which the nodes specify processing activities and arcs (lines with arrow heads) specify the data items transmitted between the processing nodes.
2) Like flowcharts, data flow diagrams can be used at any desired level of abstraction.
3) Unlike flowcharts, data flow diagrams do not indicate decision logic or conditions under which various processing nodes in the diagram might be activated.

4) They might represent data flow :-
   a) Between individual statements or blocks of statements in a routine,
   b) Between sequential routines,
   c) Between concurrent processes,
   d) Between geographically remote processing units.
5) The DFDs have basic two levels of development, they are as follows –
   3) A Level 0 DFD, also known as Fundamental System Model or Context Model, represents the entire system as a single bubble with incoming arrowed lines as the input data and outgoing arrowed lines as output.
   4) A Level 1 DFD might contain 5 or 6 bubbles with interconnecting arrows. Each process represented here is the detailed view of the functions shown in the level 0 DFD.

- Here the rectangular boxes are used to represent the external entities that may act as input or output outside the system.
- Round Circles are used to represent any kind of transformation process inside the system.
- Arrow headed lines are used to represent the data object and its direction of flow.

6) Following are some guidelines for developing a DFD:-
   7) Level 0 DFD should depict the entire software system as a single bubble.
   8) Primary input and output should be carefully noted.
   9) For the next level of DFD the candidate processes, data objects and stores should be recognized distinctively.
   10) All the arrows and bubbles should be labeled with meaningful names.
   11) Information flow continuity should be maintained in all the levels.
   12) One bubble should be refined at a time.

Example: Let us consider a software system called the root mean square (RMS) calculating system which reads 3 integers in the range from – 1000 to + 1000 and calculate their RMS value and then display it.

The context level diagram of RMS is shown below

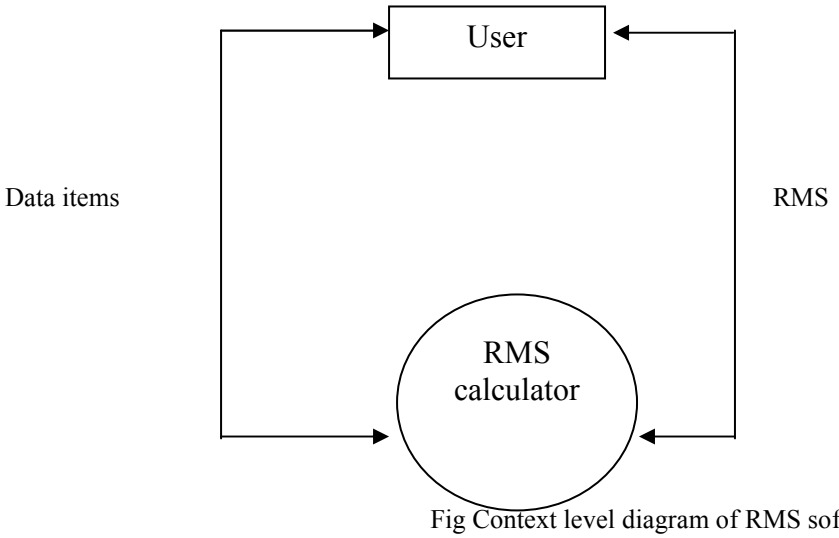


Fig Context level diagram of RMS software

And its level 1 DFD is

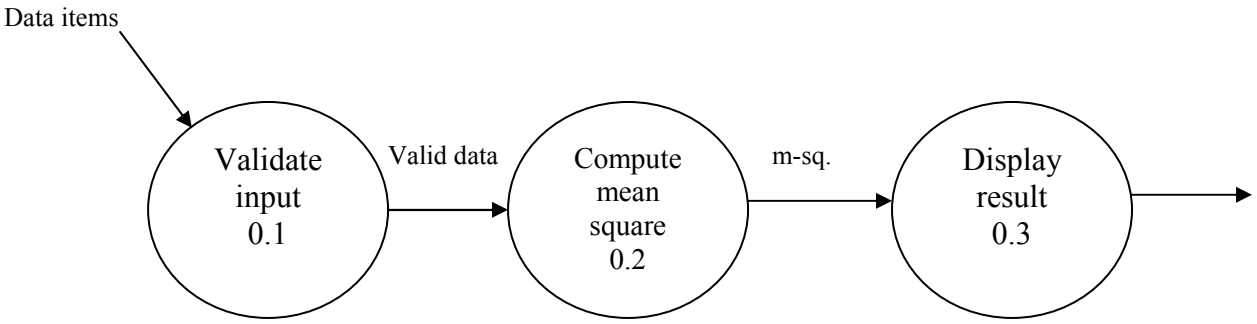


Fig Level 1 DFD of RMS Software

And its level 2 DFD is

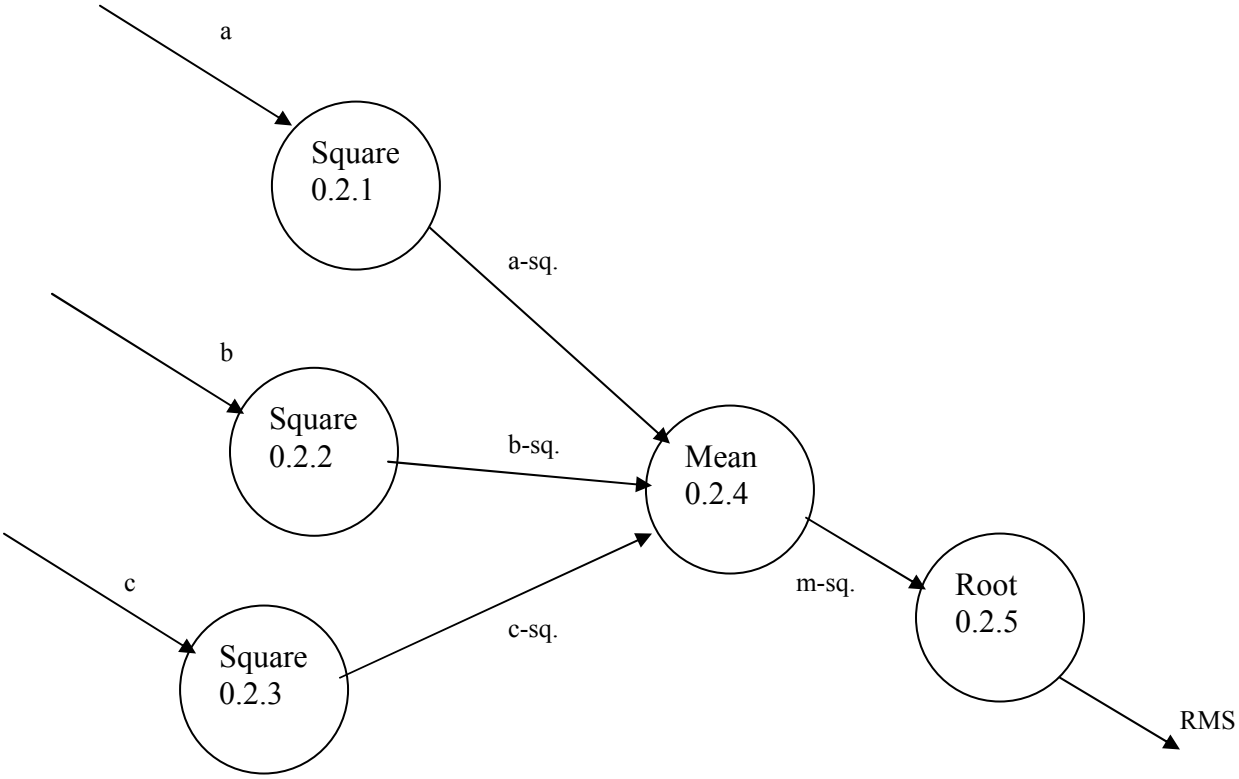
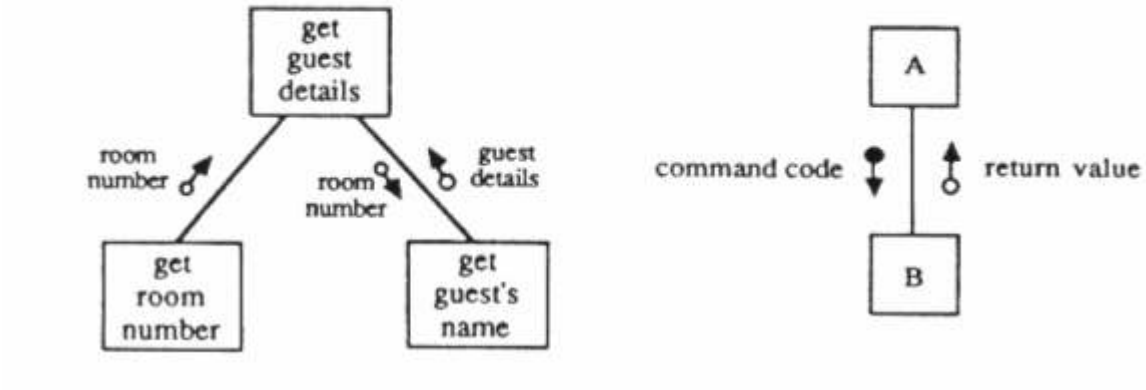


Fig Level 2 DFD of RMS software

Structure Charts

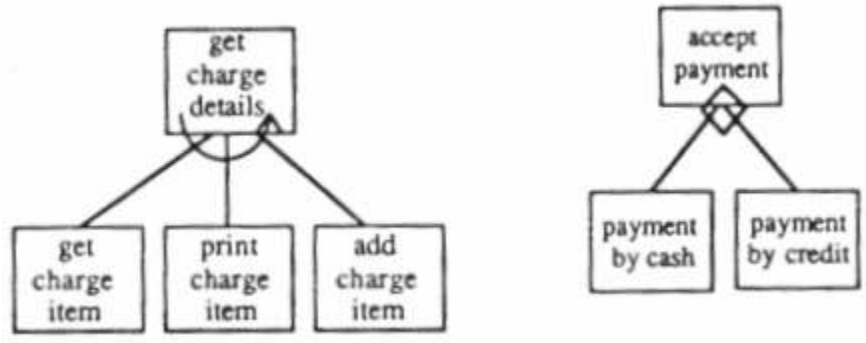
- They are used during architectural design for documenting hierarchical structure, parameters and interconnections in a system.
- In a structure chart a module is represented by a box with the module name written in the box.
- An arrow from a module A to a module B represents that the module A invokes module B. the arrow is labeled by the parameters received by B as input and the parameters returned by B.

Paremeters in Structure Chart



Repetitions and Selections:

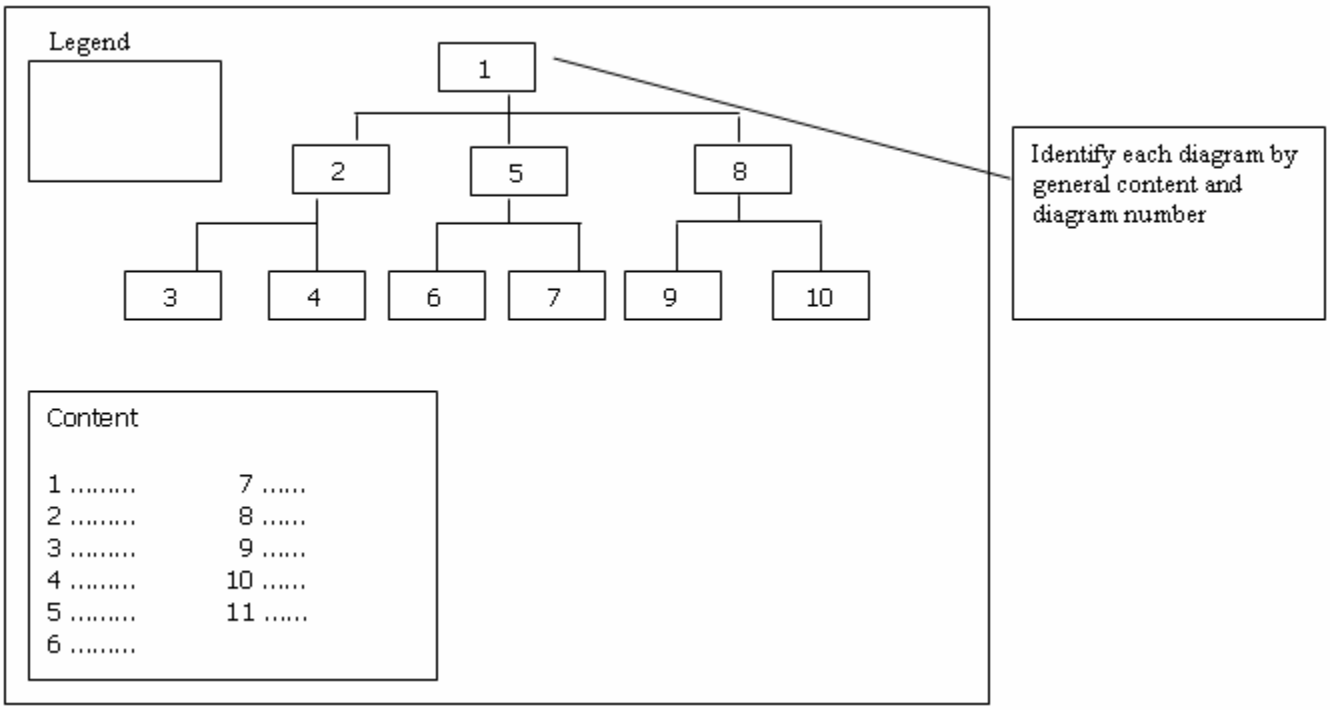
- Repetitions can be represented by a looping arrow around the arrows joining the subordinate.
- If the invocation of modules depends on the outcome of some decision it is represented by a small diamond with the arrows coming out of the diamond toward the sub-modules.



- **It differs from a flow chart in two ways –**
  a) A structure chart has no decision boxes and
  b) The elements need not be shown in a sequential order.
- Structure chart are useful to represent the model of the system however it is not useful for representing the final design as it does not give all the information needed about the design.

HIPO (Hierarchy – Process – input – Output) Diagrams:

1) It was developed at IBM labs as design representation schemes for top-down software development, and as external documentation aids for released products.
2) It contains
   a) A visual table of contents – it's a directory referencing the set of diagrams in the package, it contains a tree-structured directory, a summary of contents of each over view diagram and a legend of symbol definitions.
   b) A set of overview diagrams – they specify the functional processes in a system. Each overview diagram describes the inputs, processing steps and output for the explained function.
   c) A set of detail diagrams – which has the same format as over view diagrams.



Structured English

Structured English can be used to provide a step-by step specification for an algorithm.

It can be used at any desirable level of detail.

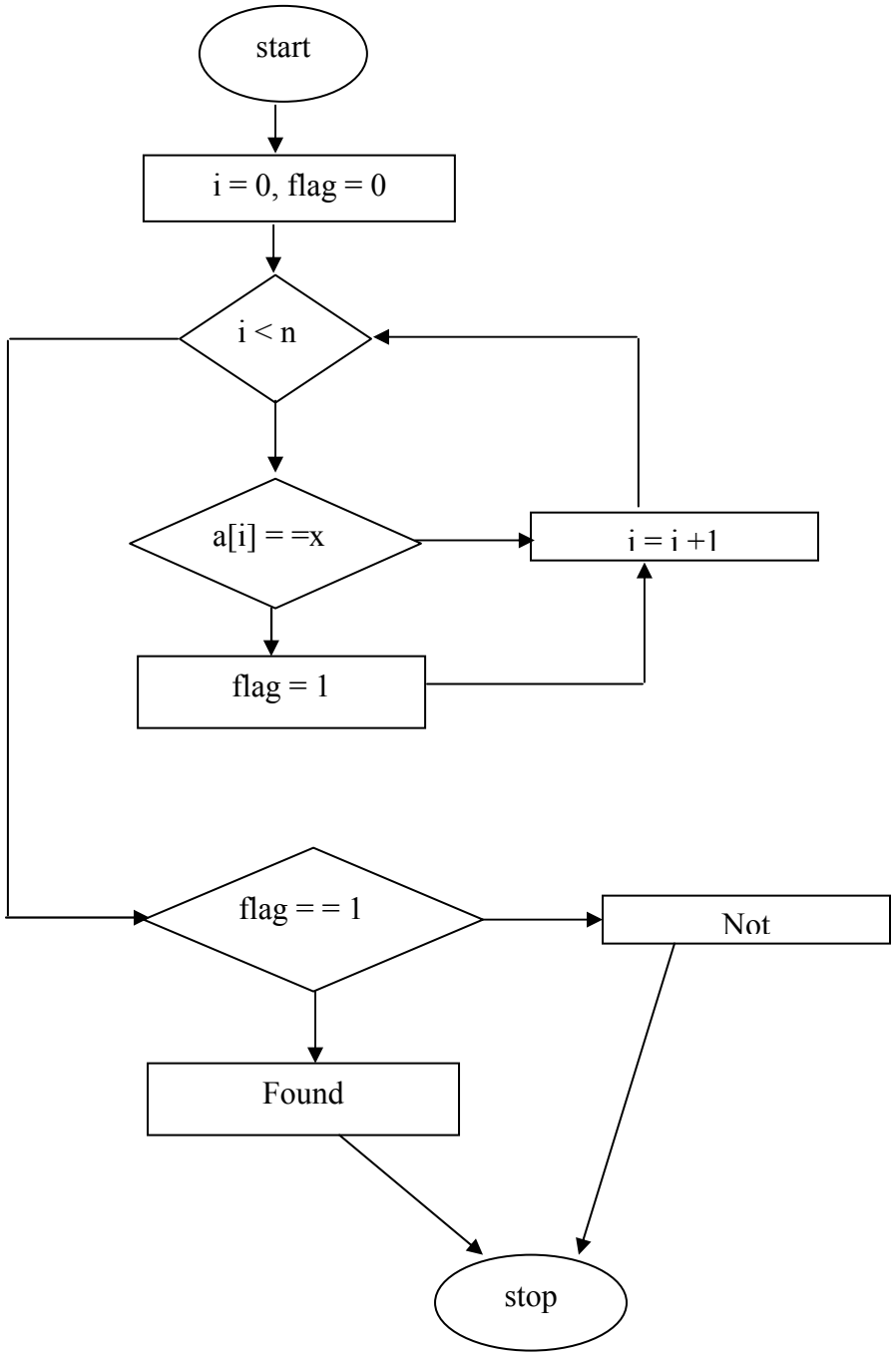
- A ***rigid subset of the English language*** omitting adjectives, adverbs, compound and complex sentences, all verb modes except imperative and most punctuation
- Result: A language containing a limited set of conditional and logic statements with nouns and strong verbs

Standards vary between organizations - objectives of: ***conciseness, preciseness*** and ***lack of ambiguity*** apply to all variants.

Structured Flowcharts:

- Flowcharts are the traditional means of specifying and documenting algorithmic details in a software system
- They are graphical representation to show the flow of information and control from element to element in a system.
- The information flow represented here is sequential.
- It is a pictorial representation of an algorithm that uses symbols to show the operations and decisions to be followed.
- They incorporate - Rectangular boxes representing some action, Diamond shaped boxes for decisions, directed arcs (arrow headed lines) for specifying interconnection and information flow between the boxes and various other graphical shapes for the representation of data stores, input, output etc.

For example
The flow chart of function search is shown below:



Decision tables

Decision tables can be used to specify complex decision logic in a high level software specification. They are also useful for specifying algorithm logic during detailed design. At this level of usage, decision tables can be specified and translated into source code logic.

Pseudocode

- Pseudocode notation can be used in both the architectural and detailed design.
- It can be used at any desired level of abstraction.
- Using pseudocode the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as If- Then-Else, While-Do and End.
- Key words and indentation describes the flow of control, while the English phrases describe processing actions.
- Pseudocode can replace flowcharts and reduce the amount of external documentation required to describe a system.
- Converting pseudocode to a programming language is much easier as compared to converting a flowchart.

Procedure Templates:

1. The format of procedure interface specification is shown below.
2. In the early stages of architectural design, only the information in level 1 need be supplied.
3. As design progress, the information on levels 2, 3 and 4 can be included in successive steps.
4. The term "side effect" in fig shown below means any effect a procedure can exert on the processing environment that is not evident from the procedure name and parameters.
5. Modifications to global variables, reading or writing a file, opening or closing a file, or calling a procedure that in turn exhibits side effects are all examples of side effects.
6. It is recommended that only the information on level 1 in fig shown below be provided during initial architectural design, because detailed specification of side effects, exception handling, processing algorithms, and concrete data representations will sidetrack the designer into inappropriate levels of detail too soon.

PROCEDURE NAME:
PART OF: (sub-system name & number) Level 1
CALLED BY:
PURPOSE:
DESIGNER/DATE(s):

PARAMETERS: (names, modes, attributes, purposes)
INPUT ASSERTION: (preconditions) Level 2
OUTPUT ASSERTION: (postconditions)
GLOBALS: (names, modes, attributes, purposes, shared with)
SIDE EFFECTS:

LOCAL DATA STRUCTURES: (names, attributes, purposes)
EXCEPTIONS: (conditions, responses) Level 3
TIMING CONSTRAINTS:
OTHER LIMITATIONS:

PROCEDURE BODY: (pseudocode, structured English, structured flowchart, decision table) Level 4

Fig Format of the procedure template

Mapping Requirements into Software Architecture

There are two types of information flow which are the drivers for the mapping approach:

a) Transform Flow.
b) Transaction Flow.

Transform Flow – Information must enter and exit software in an "external world" form. External data (like data typed on keyboard, tones on a telephone line and video images in multimedia application) must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as incoming flow. At the kernel of the software, a transition occurs. Incoming data are passed through a 'transform center' and begin to move along paths that now lead "out" of the software. Data moving along these paths are called outgoing flow. The overall flow of data occurs in a sequential manner and follows straight line paths. When a segment of a DFD exhibits these characteristics, 'transform flow' is present.

Transaction Flow – The fundamental system model implies transaction flow; therefore, it is possible to characterize all data flow by a single data item, called a transaction that triggers other data flow along one of the many paths. So, when DFD takes the form as shown below, transaction flow is present.

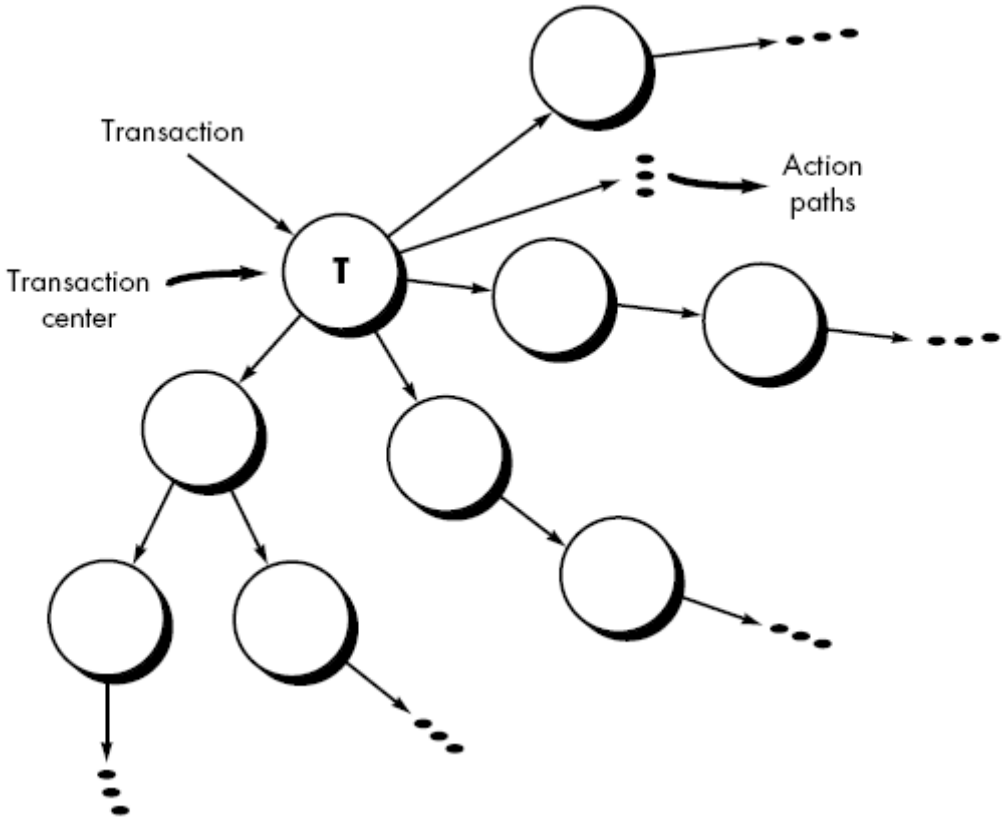


Fig Transaction Flow

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and based on its value; flow along one of many action paths is initiated. "**The hub of information flow from which many actions path originate is called a transaction center**".

It should be note that, within a DFD for a large system, both transform flow and transaction flow may be present. For example, in a transaction – oriented flow, information flow along an action path may have transform flow characteristics.

Transform Mapping – "Transform Mapping is a set of design steps that allow a DFD with transform flow characteristics to be mapped into a specific architectural style".

Structural design provides 2 strategies to guide transformation of a DFD into a structure chart:

a) Transform Analysis
b) Transaction Analysis

Transform Analysis (Transaction mapping) – It identifies the primary functional components (modules) and the high level input and output for these components.

First step in transform analysis is to divide the DFD into three parts:

a) Input
b) Logical processing
c) Output

The input portion in the DFD includes processes that transform input data from physical (e.g., character from terminal) to logical form (e.g., internal tables, lists etc). Each input portion is called '**afferent branch**'. There may be more than one afferent branch in a DFD.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called on '**efferent branch**'. The remaining portion of a DFD is called **central transform**.

In second step of transform analysis, the structure chart is derived by drawing one functional component for each central transform, for each afferent and efferent branch. Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input of filter data from it are central transforms. The first level of structure chart is produced by representing each input and output unit as boxes and each transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding sub– functions required by each of the high level functional components. Many levels of functional components may be added. **This process of breaking functional components into subcomponents is called 'factoring'**. Factoring includes adding, read and write modules, error handling modules, initialization and termination processes. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example: Let us consider a software system called the root mean square (RMS) calculating system which reads 3 integers in the range from – 1000 to + 1000 and calculate their RMS value and then display it. Apply transform analysis to design its structure chart.

Sol:

Let us consider the DFD of RMS

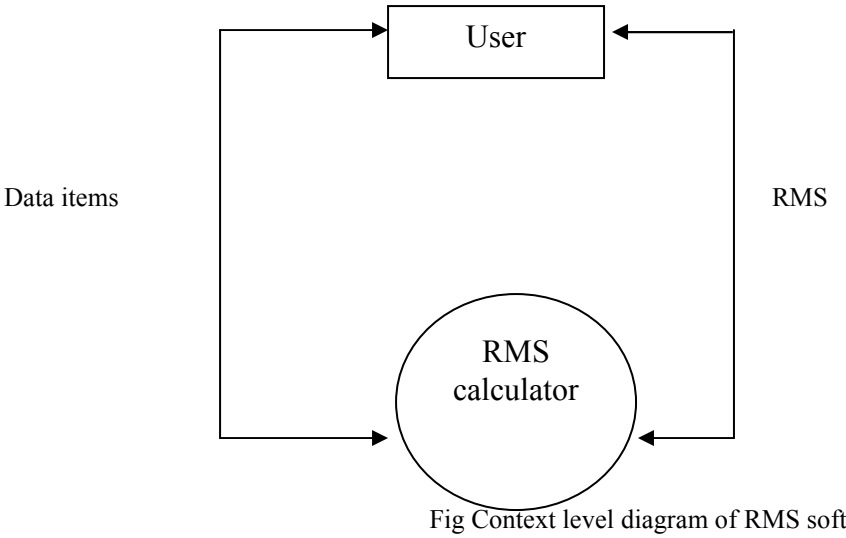


Fig Context level diagram of RMS software

And its level 1 DFD is

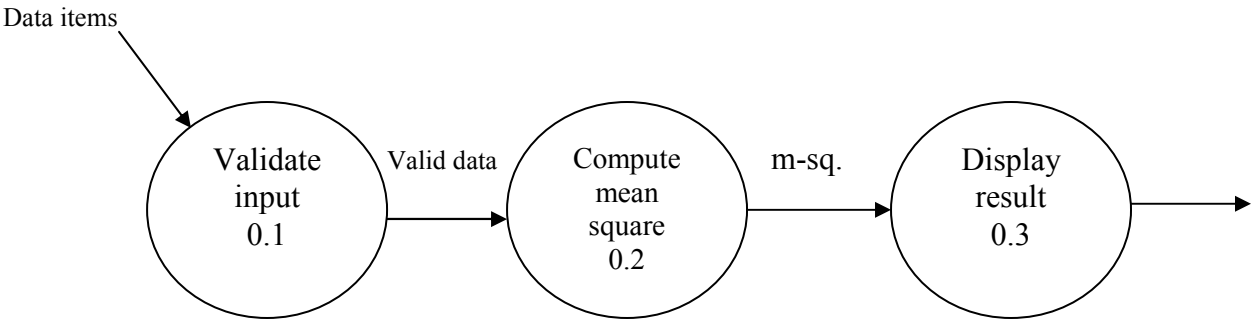


Fig Level 1 DFD of RMS Software

And its level 2 DFD is

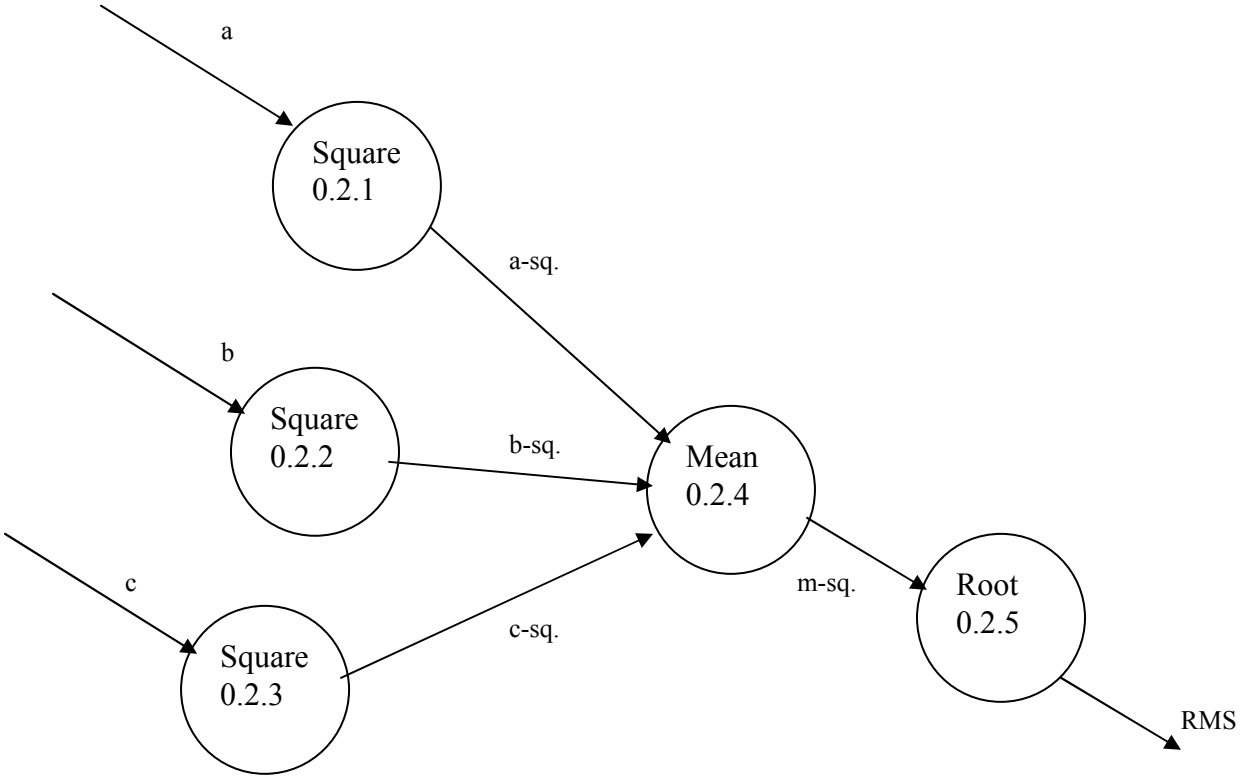


Fig Level 2 DFD of RMS software

By observing, level 1 DFD, we can identify the read input and validate input as the afferent branch and the write output as the efferent branch. By applying the 2nd and 3rd steps of transform analysis, we get the structure chart below

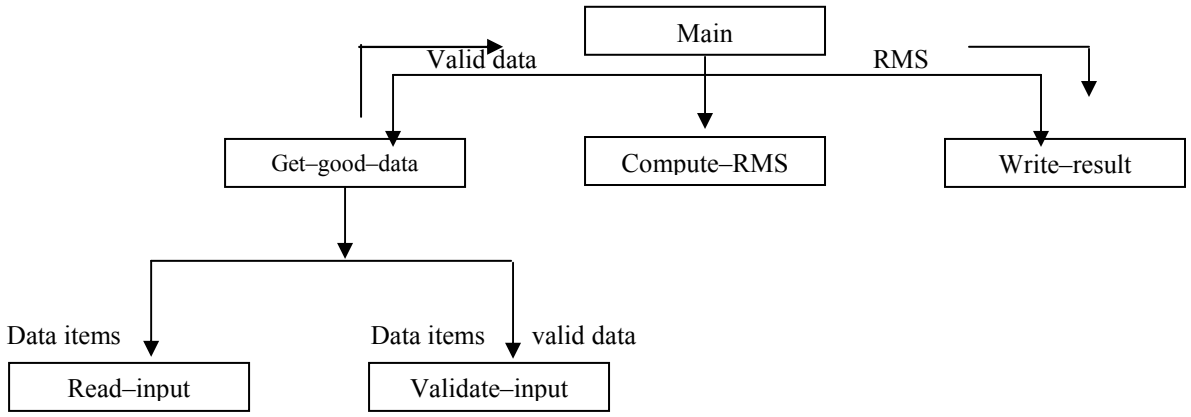


Fig Structure chart of RMS software

Transaction analysis (Transaction mapping) – It is an alternative to transform analysis and is useful while designing transaction processing programs. A transform centered system is characterized by similar processing steps for each data item processed by input, process and output systems. In a transaction driven system, one of several possible paths through the DFD is traversed depending upon the input data item.

A transaction is any element of data that triggers an action. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction center module. This is an architectural design.

User Interface Design

Interface design focuses on three areas of concern:

a) The design of interfaces between software components,
b) The design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and
c) The design of the interface between a human (i.e., the user) and the computer.

User Interface Design Principles

Theo Mandel coins three "golden rules" for user interface design:

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

1. Place the user in control

During a requirements gathering session for a major new information system, a key user was asked about the attributes of the window oriented graphical interface. Mandel defines a number of design principles that allow the user to maintain control:

a) Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
b) Provide for flexible interaction.
c) Allow user interaction to be interruptible and undoable.
d) Streamline interaction as skill levels advance and allow the interaction to be customized.
e) Hide technical internals from the casual user.
f) Design for direct interaction with objects that appear on the screen.

2. Reduce the user's memory load

The more a user has to remember, the more error-prone will be the interaction with the system. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel defines design principles that enable an interface to reduce the user's memory load:

a) Reduce demand on short – term memory.
b) Establish meaningful defaults.
c) Define shortcuts that are intuitive.
d) The visual layout of the interface should be based on a real world metaphor.
e) Disclose information in a progressive fashion.

3. Make the interface consistent

The interface should present and acquire information in a consistent fashion. This implies that

(1) All visual information is organized according to a design standard that is maintained throughout all screen displays,
(2) Input mechanisms are constrained to a limited set that are used consistently throughout the application, and
(3) Mechanisms for navigating from task to task are consistently defined and implemented.

Mandel defines a set of design principles that help make the interface consistent:

a) Allow the user to put the current task into a meaningful context.
b) Maintain consistency across a family of applications.
c) If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

The User Interface Design Process

The design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure below, the user interface design process encompasses four distinct framework activities:

a) User, task, and environment analysis and modeling
b) Interface design
c) Interface construction
d) Interface validation

The spiral shown in Figure below implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the implementation activity involves prototyping—the only practical way to validate what has been designed.

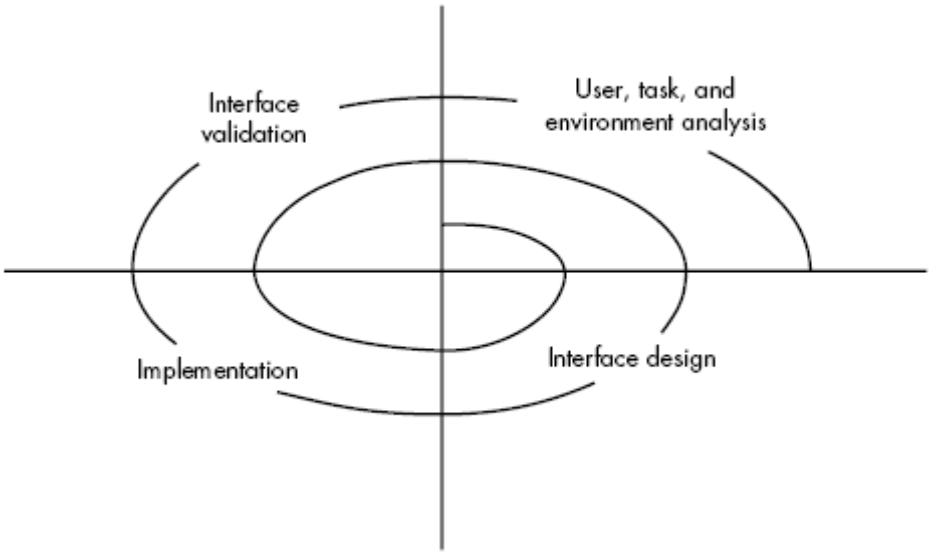


Fig User Interface Design Process

a) User, task, and environment analysis and modeling

The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, the software engineer attempts to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

b) Interface design

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

c) Interface construction

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

d) Interface validation

Validation focuses on

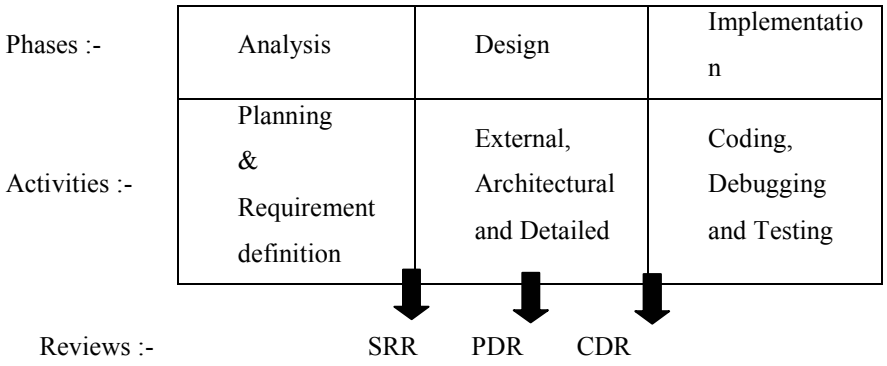
1. The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;
2. The degree to which the interface is easy to use and easy to learn; and
3. The users' acceptance of the interface as a useful tool in their work.

Milestones, Walkthroughs and Inspections

These activities help in exposing errors, provide increased project communication, keeping the project in schedule, and verification that the design satisfies the requirements.

Milestones

1) These are a set of occasions in project design where the proper progress of the project can be assessed in such a way that corrective measures could be taken if necessary.
   a) The two major milestones are –
      i. Preliminary Design Review (PDR) :- Its normally held near the end of architectural design and prior to detailed design
      ii. Critical design Review (CDR) :- Its normally held at the end of detailed design and prior to implementation.
   b) The major goal of PDR is to demonstrate the externally observable characteristics and architectural structure of the product which would satisfy the customer's requirements. Functional characteristics, performance attributes, external interface, user dialogs, report formats, exception conditions and exception handling and future enhancements are reviewed during PDR.
   c) The CDR provides a final management decision point , to build or cancel the system.



SRR:- Software Requirements Review

PDR: - Preliminary Design Review.

CDR: - Critical Design Review.

Walkthroughs

2) A structured walkthrough is an in-depth, technical review of some aspects of a software system. Walkthroughs can be anytime, during any phase of a software project.
3) A walkthrough team consists of 4 to 6 people. The person whose material is being reviewed is responsible for providing copies of the review materials to the members of the walkthrough group in advance of the walkthrough session and the team members are responsible for understanding the reviewing material before the session.
4) During the walkthrough the reviewed "walks through" the material while the reviewers look for errors, request clarification and explore problem areas in the material under review.
5) High-level managers should not attend walkthrough sessions as the aim of walkthroughs is error detection not corrective action. Its important to note that the material is reviewed not the person whose material is being reviewed.

Inspections

1) Design inspections are conducted by teams of trained inspectors who have a check list of items to be examined.
2) Special forms are used to record problems encountered.
3) A typical inspection team consists of a Moderator or Secretary, a Designer, an Implementor and a Tester. The Designer, Implementor and Tester may or may not be the people responsible for the actual design, implementation and testing of the product being inspected.
4) The team members are trained for their specific roles and typically conduct a dual 2-hrs sessions per day.

Software Configuration Management

1. ***Software configuration management*** **(SCM)** is an umbrella activity that is applied throughout the software process, because change can occur at any time.
2. SCM activities are developed to
   (1) Identify change,
   (2) Control change,
   (3) Ensure that change is being properly implemented, and
   (4) Report changes to others who may have an interest.
3. It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that begin when a software engineering project begins and terminate only when the software is taken out of operation.
4. A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made.
5. The output of the software process is information that may be divided into three broad categories:
   (1) Computer programs (both source level and executable forms);
   (2) Documents that describe the computer programs (targeted at both technical practitioners and users), and
   (3) Data (contained within the program or external to it).
   The items that comprise all information produced as part of the software process are collectively called a ***software configuration***.
6. As the software process progresses, the number of ***software configuration items*** **(SCIs)** grows rapidly. A ***System Specification*** spawns a ***Software Project Plan*** and ***Software Requirements Specification*** (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs, little confusion would result. Unfortunately, another variable enters the process—***change***. Change may occur at any time, for any reason. In fact, the First Law of System Engineering states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."
7. There are four fundamental sources of change:
   a) New business or market conditions dictate changes in product requirements or business rules.
   b) New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
   c) Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
   d) Budgetary or scheduling constraints cause a redefinition of the system or product.
8. Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process.

Baselines

A ***baseline*** is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE defines a baseline as: "A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

One way to describe a baseline is through analogy:

Consider the doors to the kitchen in a large restaurant. One door is marked OUT and the other is marked IN. The doors have stops that allow them to be opened only in the appropriate direction.

If a waiter picks up an order in the kitchen, places it on a tray and then realizes he has selected the wrong dish, he may change to the correct dish quickly and informally before he leaves the kitchen.

If, however, he leaves the kitchen, gives the customer the dish and then is informed of his error, he must follow a set procedure:

(1) Look at the check to determine if an error has occurred,

(2) Apologize profusely,

(3) Return to the kitchen through the IN door,

(4) Explain the problem, and so forth.

A baseline is analogous to the kitchen doors in the restaurant. Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, we figuratively pass through a swinging oneway door. Changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review. For example, the elements of a ***Design Specification*** have been documented and reviewed. Errors are found and corrected. Once all parts of the specification have been reviewed, corrected and then approved, the ***Design Specification*** becomes a baseline. Further changes to the program architecture (documented in the ***Design Specification***) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure below.

The progression of events that lead to a baseline is also illustrated in Figure below. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a ***project database*** (also called a ***project library*** or ***software repository***). When a member of a software engineering team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private work space. However, this extracted SCI can be modified only if SCM controls are followed. The arrows in Figure above illustrate the modification path for a baselined SCI.

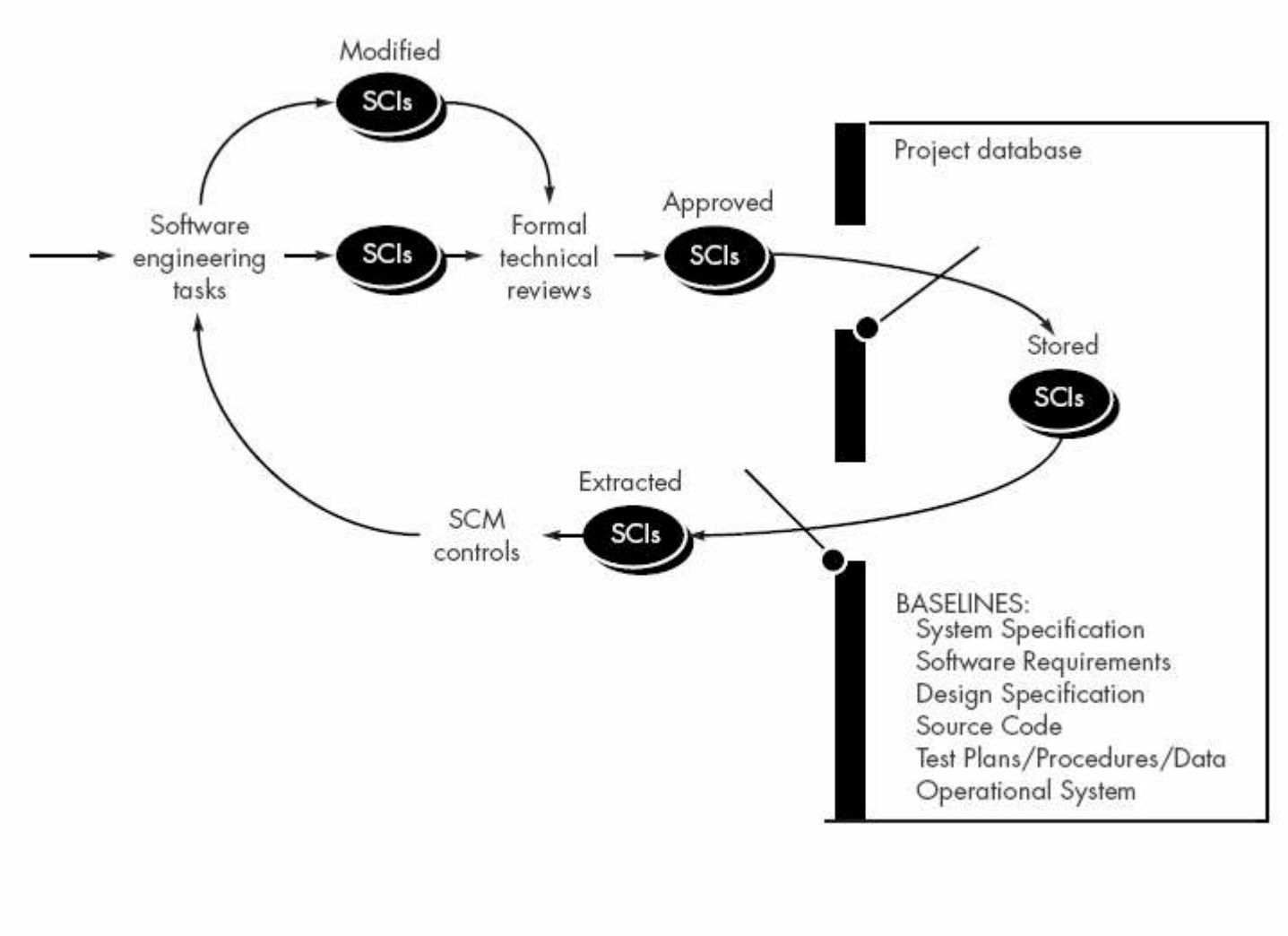


Fig Baselined SCIs and the project database

Software Configuration Items

1. A SCI could be considered to be a single section of a large specification or one test case in a large suite of tests.
2. An SCI is a document, an entire suite of test cases, or a named program component (e.g., a C++ function or an Ada package).
3. In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, and other CASE tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made.
4. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process. In reality, SCIs are organized to form ***configuration objects*** that may be cataloged in the project database with a single name.
5. A configuration object has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure below, the configuration objects, **Design Specification, data model, component N, source code** and **Test Specification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a ***compositional relation***. That is, **data model** and **component N** are part of the object **Design Specification.** A double-headed straight arrow indicates an interrelationship.
6. If a change were made to the source code object, the interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.

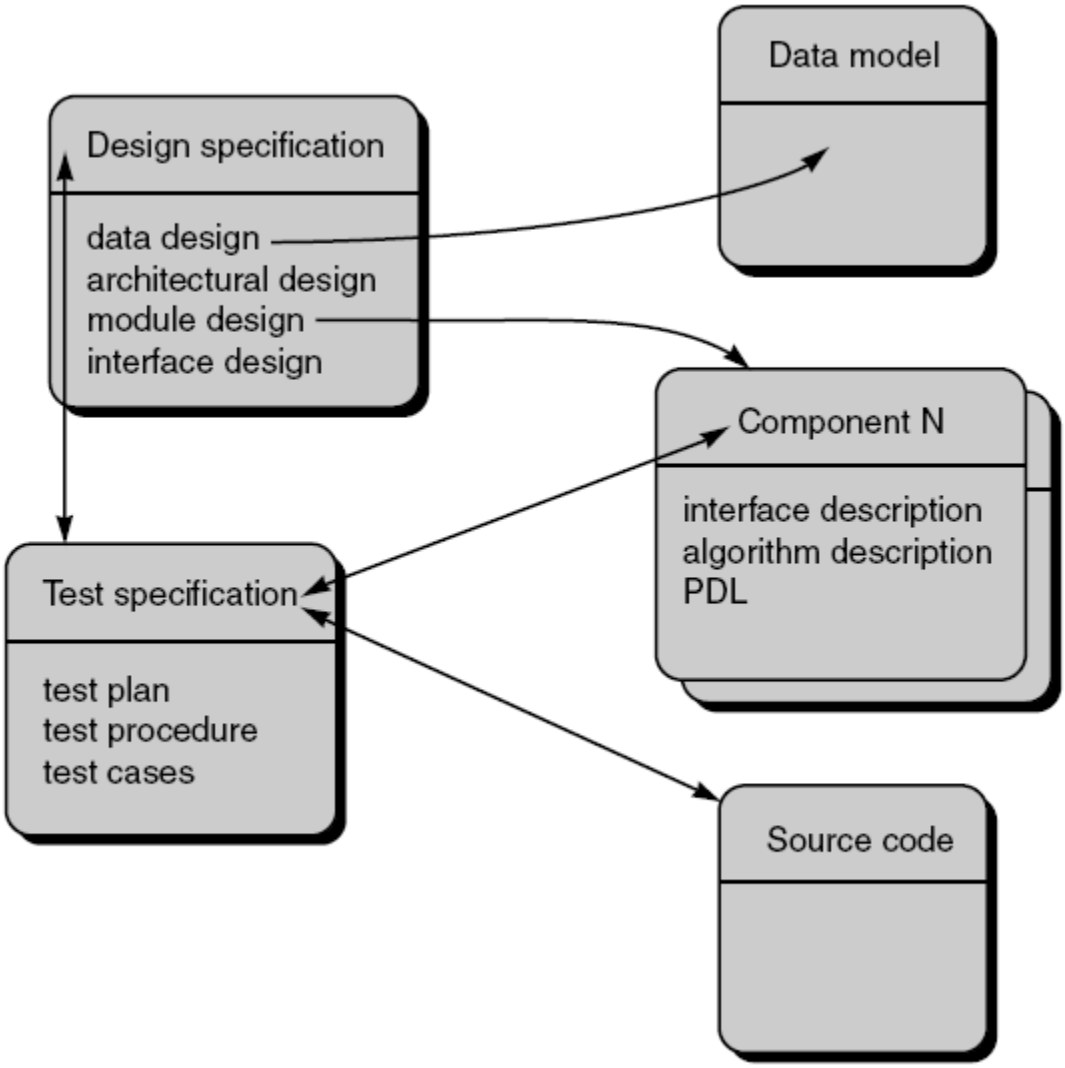


Fig Configuration objects

SCM Process

The five basic SCM tasks are

i) Identification of objects,
ii) Version control,
iii) Change control,
iv) Configuration auditing and
v) Reporting.

1) **Identification of objects –** To control and manage SCIs, each of them must be separately named and then organized using object-oriented approach. The SCIs can be classified in to two kinds of objects –
   a) Basic objects and
   b) Aggregate objects.

   a) **Basic object –** It's a "unit of text" created by a software engineer during analysis, design, coding or testing phases. For example, a basic object might be a section of a requirements specification, a source listing for a component, or a suite of test cases that are used to exercise the code.
   b) **Aggregate object –** It's a collection of basic objects and other aggregate objects. For example, Design Specification is an aggregate object.

Each object has a set of unique features for distinct identification. Every object should have –

i. An unique name,
ii. A description of the object including its SCI type eg. Document, program or data, a project identifier and version information,
iii. A list of resources, which are entities that are processed, provided or required by the object.

2) **Version control** – It combines various tools and procedures to manage and control different versions of SCI objects (as a result of change) that are created during the software engineering process. **Clemm** describes version control in the context of SCM:

"Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes".

These "attributes" mentioned can be as simple as a specific version number that is attached to each object or as complex as a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system.

3) **Change control –** For a large system development project, uncontrolled change rapidly leads to confusion and inconsistencies. Change control includes human procedures and automated tools to control the reasons for change. The following processes take place when a situation of change occurs:
   i. **Change request –** A change request is first submitted and then it is evaluated to asses its –
      i) Technical merit,
      ii) Potential side effects, subsystems and the cost for implementing the change.
   ii. **Change report** – After the evaluation is done, a change report is created that is submitted to the Change Control Authority/Board (CCA or CCB). CCA or CCB is a group who are responsible for evaluating the change report and makes the final decision on the status and priority of the change. This group generates an Engineering Change Order (ECO) for each approved change.
   iii. **ECO (Engineering Change Order) –** It consists of
      i) The description of the change to be made,
      ii) Constraints that has to be taken care of and
      iii) Criteria for review and audit.
   iv. **Check out & check in –** The object to be changed is "checked out" of the project database, the decided changes are made and appropriate SQA (Software Quality Assurance) activities are performed. The object is then "checked in" the project database and appropriate version control mechanisms are used to create the next version of the software.

4) **Formal technical reviews (FTR) & Configuration audit –** These two activities are required to ensure that the change made to the software is properly implemented.
   a) **Formal technical reviews (FTR) –** It's a part of Software Quality Assurance (SQA) procedures. The objectives of FTR are
      i) To uncover errors in functions, logic or implementation,
      ii) To verify that the software under review should meet its requirement,
      iii) To ensure that the representation of the software is according to the standards,
      iv) To make the project more manageable
      v) It consists of walkthroughs, inspections and round-robin reviews.

b) **Software configuration audit** – It consists of the following auditing procedures

i) To check whether the changes specified in the ECO (Engineering Change Order) has been properly made and to check if any additional modules are added,

ii) To check whether formal technical reviews are conducted for the assessment of technical correctness,

iii) To check whether SE standards are properly followed,

iv) To check whether the change is highlighted in the SCI (Software Configuration Items) documentation and also the attributes of the configuration object should reflect the change,

v) To check whether SCM (Software Configuration Management) procedures like noting change, recording it and reporting it has been properly done,

vi) To check whether all the related SCIs are properly updated.

5) **Configuration Status reporting (CSR) –** Its an SCM task which summarizes the activities done so far which includes the following

a) The report of all the activities done,

b) The report on the persons involved in the above reported activities,

c) The report on the time and date when the reported activities were accomplished,

d) The report on various other factors or objects that will be affected due to the reported activity.

Following are the situations where the CSR needs updating

a) Each time when a SCI is assigned a new or updated identification,

b) Each time ECO is issued i.e a when a change is approved by the CCA/CCB,

c) Each time a configuration audit is conducted.

The CSR is made available online and is updated occasionally in order to keep the management and concerned individuals updated on the changes. It improves the communication among all the people involved.

Unit 4

Software Testing

Introduction

Once the source code has been developed, testing is required to uncover the errors before it is implemented. In order to perform software testing a series of test cases is designed. Since testing is a complex process hence, in order to make the process simpler, testing activities are broken into smaller activities. Due to this, for a project incremental testing is generally preferred. In this testing process the system is broken in set of subsystems and there subsystems are tested separately before integrating them to form the system for system testing.

Definition of Testing

1. **According to IEEE** – "Testing means the process of analyzing a software item to detect the differences between existing and required condition (i.e. bugs) and to evaluate the feature of the software item".
2. **According to Myers** – "Testing is the process of analyzing a program with the intent of finding an error".

Primary objectives of Software Testing

According to Glen Myers the primary objectives of testing are:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding and as yet undiscovered error.
3. A successful test is one that uncovers as yet undiscovered error.

Testing cannot show the absence of errors and defects, it can show only error and detects present. Hence, objective of testing is to design tests that systematically uncover different errors and to do so with a minimum amount of time and effort.

ERRORS, FAULT AND FAILURE

Error – The term error is used to refer to the discrepancy between computed, observed or measured value and the specified value. In other terms errors can be defined as the difference between actual output of software and correct output.

Fault – It is a condition that causes a system to fail in performing its required function.

Failure – A software failure occurs if the behavior of the software is different from specified behavior. It is a stage when system becomes unable to perform a required function according to the specification mentioned.

TEST ORACLES

A test oracle is a mechanism, different from the program itself that can used to check the correctness of the output of the program for the test cases. In order to test any program we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this test oracle is needed.

Test oracles generally use the system specification of the program to decide what the correct behavior of the program should do. In order to determine the correct behavior it is important that the behavior of the system be unambiguously specified and the specification itself should be error free.

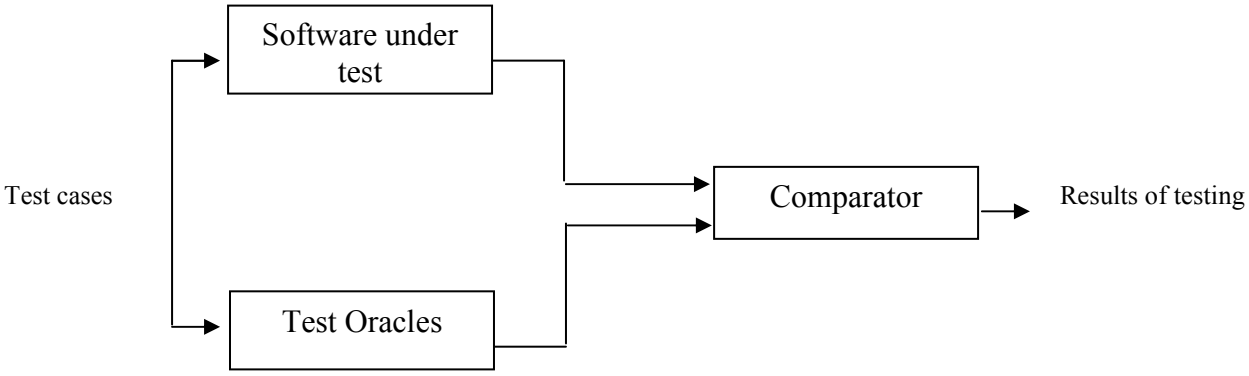


Fig (a) Test Oracles

Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggested a set of testing principles as follows

1. **All tests should be traceable to customer requirements –** According to customer's point of view the most severe defects are those that cause the program to fail to meet its requirements. Hence, the tests should be done in keeping the objective of customers requirement is must.
2. **Tests should be planned long before testing begins –** Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
3. **The Pareto principle applies to software testing –** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
4. **Testing should begin "in the small" and progress toward testing "in the large" –** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
5. **Exhaustive testing is not possible –** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
6. **To be most effective, testing should be conducted by an independent third party –** By most effective, we mean testing that has the highest probability of finding errors. The software engineer who created the system is not the best person to conduct all tests for the software, so to be more effective third independent party should be involved on performing test.

Attributes of Good Test

Kaner, Falk, and Nguyen suggest the following attributes of a "good" test:

1. **A good test has a high probability of finding an error.**

   To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

2. **A good test is not redundant.**

   Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). For example, a module of software is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester

designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.

3. **A good test should be "best of breed".**

   In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

4. **A good test should be neither too simple nor too complex.**

   Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

Test Case Design

The test case design provides the developer with a systematic approach to test and also provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood errors in software.

Since, effort and time both is needed to execute the test cases and also machine time is needed to execute the program for those test cases also, effort is needed to evaluate the results. Hence, the fundamental goals of a testing activity are to minimize the test cases and maximize the error unfolded. According to Yamaura there is only one rule in designing test cases and it is that it should cover all features, but do not make too many test cases.

An ideal set of test cases is one that includes all the possible inputs to the program. This is called **exhaustive testing**. However, this testing is practically not possible because number of elements in the input domain is very large and testing them through one ideal set is not feasible. Hence, realistic goal for testing is to select a set of test cases that is close to the ideal.

White Box Testing (Gloss Box Testing) (Structural Testing)

1. In this testing technique the internal logic of software components is tested.
2. It is a test case design method that uses the control structure of the procedural design test cases.
3. It is done in the early stages of the software development.
4. Using this testing technique software engineer can derive test cases that:
   a) All independent paths within a module have been exercised at least once.
   b) Exercised true and false both the paths of logical checking.
   c) Execute all the loops within there boundaries.
   d) Exercise internal data structures to ensure their validity.

Advantages:

1. As the knowledge of internal coding structure is prerequisite, it becomes very easy to find out which type of input/data can help in testing the application effectively.
2. The other advantage of white box testing is that it helps in optimizing the code.
3. It helps in removing the extra lines of code, which can bring in hidden defects.
4. We can test the structural logic of the software.
5. Every statement is tested thoroughly.
6. Forces test developer to reason carefully about implementation.
7. Approximate the partitioning done by execution equivalence.
8. Reveals errors in "hidden" code.

Disadvantages:

1. It does not ensure that the user requirements are fulfilled.
2. As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, which increases the cost.
3. It is nearly impossible to look into every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.
4. The tests may not be applicable in real world situation.
5. Cases omitted in the code could be missed out.

Black Box Testing (Behavioral Testing) (Closed box Testing)

1. This technique exercises the input and output domain of the program to uncover errors in program, function, behavior and performance.
2. Software requirements are exercised using "black box" test case design technique.
3. It is done in the later stage of the software development.
4. Black box testing technique attempts to find errors related to:
    a) Missing functions or incorrect functions.
    b) Errors created due to interfaces.
    c) Errors in accessing external databases.
    d) Performance related errors.
    e) Behavior related errors.
    f) Initialization and termination errors.
5. In Black box testing the main focus is on the information domain. Test is designed for following questions:
    a) How is functionality validation testing?
    b) What class of input will make good test cases?
    c) Is the system particularly sensitive to certain input values?
    d) How are the boundaries of a data class isolated?
    e) What data rates and data volume can the system tolerate?
    f) What effects will specific combinations of data have on system operation?

Advantages:

1. More effective on larger units of code than glass box testing.
2. Tester needs no knowledge of implementation, including specific programming languages.
3. Tester and programmer are independent of each other.
4. Tests are done from a user's point of view.
5. Will help to expose any ambiguities or inconsistencies in the specifications.
6. Test cases can be designed as soon as the specifications are complete.

Disadvantages:

1. Only a small number of possible inputs can actually be tested, to test every possible input stream would take nearly forever.
2. Without clear and concise specifications, test cases are hard to design.
3. There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried.
4. May leave many program paths untested.
5. Cannot be directed toward specific segments of code which may be very complex (and therefore more error prone).
6. Most testing related research has been directed toward glass box testing.

Difference between Black box and White box testing

No.	Black Box Testing	White Box Testing
1	This method focuses on functional requirements of the software i.e., it enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a system.	This method focuses on procedural details i.e., internal logic of a program.
2	It is not an alternative approach to white box technique rather is complementary approach that is likely to uncover a different class of errors.	It concentrates on internal logic, mainly.
3	Black box testing is applied during later stages of testing.	White box testing is performed early in the testing process.
4	It attempts to find errors in following categories: (a) incorrect or missing functions. (b) interface errors. (c) errors in data structures or external database access. (d) performance errors. (e) initialization and termination errors.	White box testing attempts errors in following cases: (a) internal logic of your program (b) status of program.
5	It disregards control structure of procedural design (i.e., what is the control structure of our program, we do not consider here).	It uses control structure of the procedural design to derive test cases.
6	Black box testing, broadens our focus on the information domain and might be called as "testing in the large" i.e., testing bigger monolithic programs.	White box testing is "testing in small" i.e., testing small program components (e.g. modules or small group of modules).
7	Using black box testing techniques, we derive a set of test cases that satisfy following criteria: (a) Test cases that reduce, (by a count that is greater than 1), the number of additional test cases that must be designed to achieve reasonable testing. (b) and test cases that tell us something about the presence or absence of classes of errors rather than an error associated only with the specific tests at hand.	Using White box testing, the software engineer can derive test cases that: (a) guarantee that all independent paths within a module have been exercised at least once. (b) exercise all logical decisions on their true and false sides. (c) execute all loops at their boundaries and within their operational bounds. (d) and exercise internal data structures to ensure their validity.
8	It includes the tests that are conducted at the software interface	A close examination of procedural detail is done.
9	Are used to uncover errors.	Logical paths through the software are tested by providing test cases, that exercise specific sets of conditions or loops.
10	To demonstrate that software functions are operational i.e., input is properly accepted and output is correctly produced. Also, the integrity of external information (e.g. a data base) is maintained.	A limited set of logical paths be however examined.

Basic Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure below. Each structured construct has a corresponding flow graph symbol.

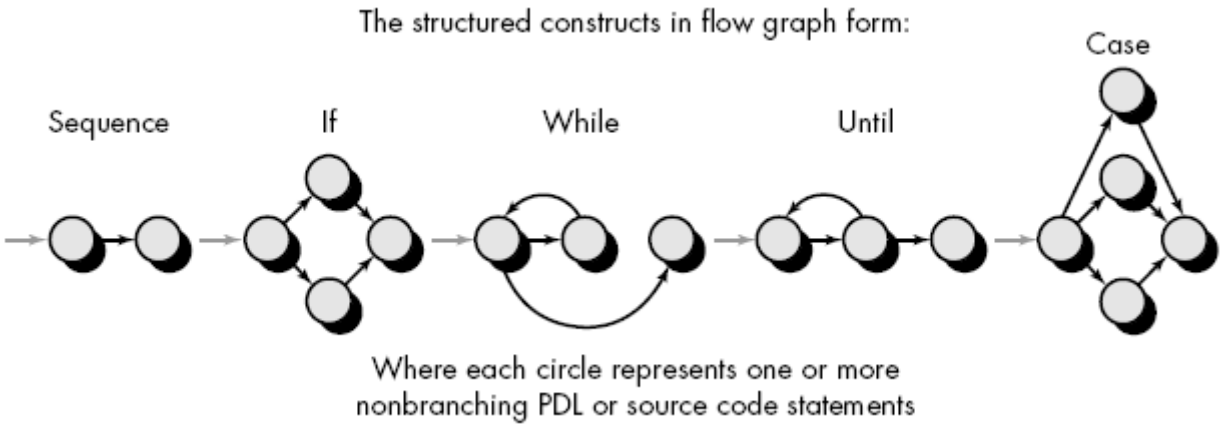


Fig (b) Flow Graph Notation

To illustrate the use of a flow graph, we consider the procedural design representation in Figure (c) below. Here, a flowchart is used to depict program control structure.

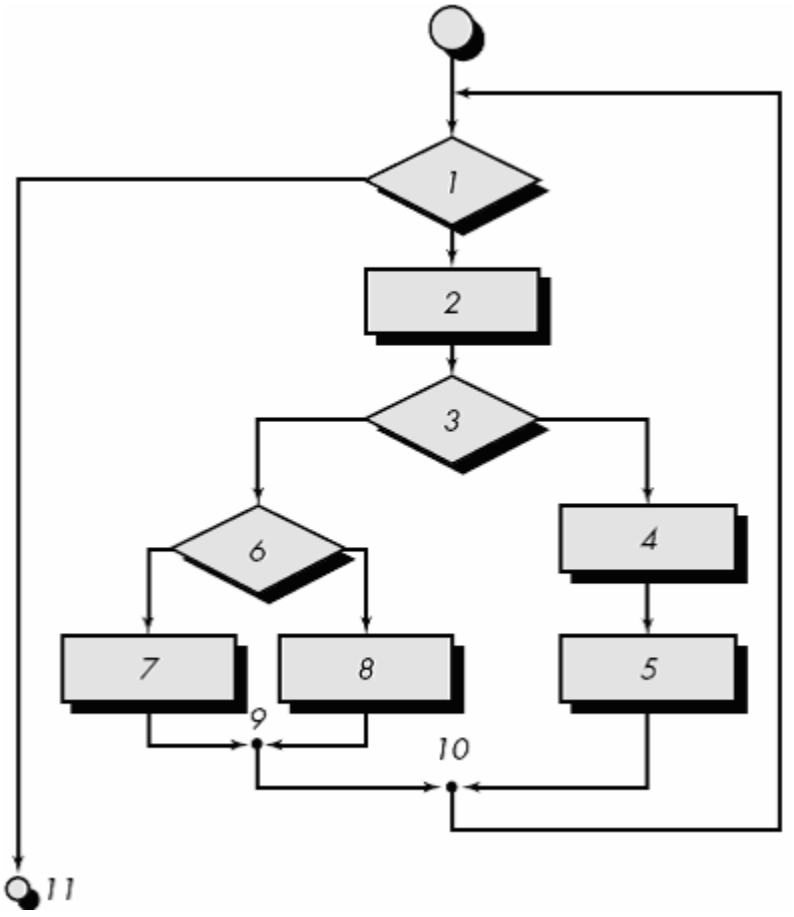


Fig (c) Flow Chart

Figure (d) maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).

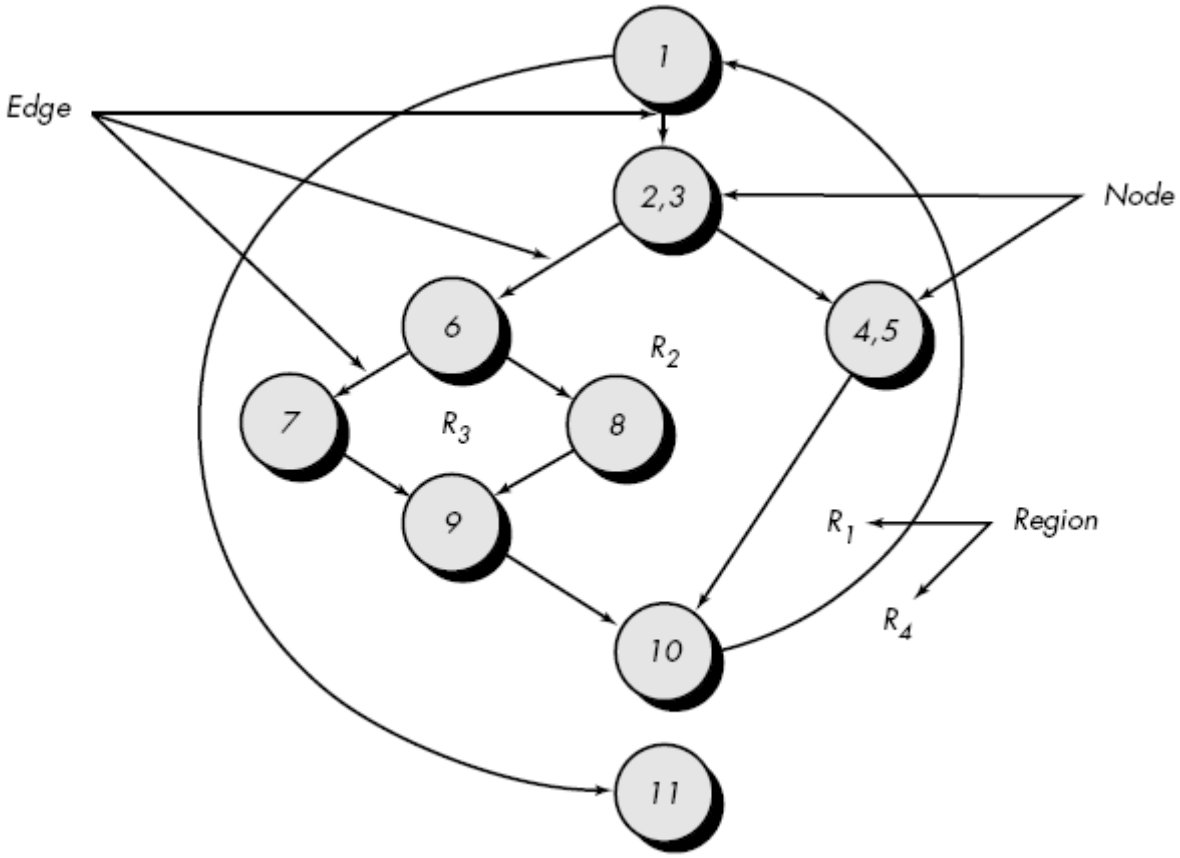


Fig (d) Flow Graph

Referring to Figure above, each circle, called a ***flow graph node,*** represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called ***edges*** or ***links,*** represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region.

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure (e) below, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a ***predicate node*** and is characterized by two or more edges emanating from it.

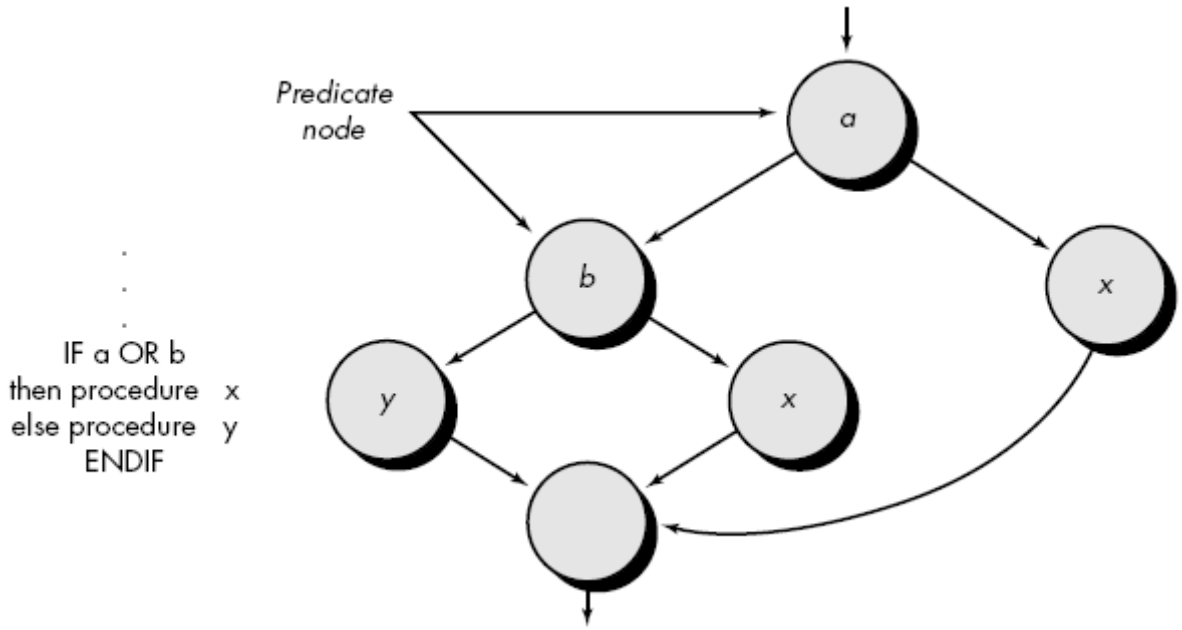


Fig (e) Compound logic

Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An ***independent path*** is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure (d) is

Path 1: 1–11

Path 2: 1–2–3–4–5–10–1–11

Path 3: 1–2–3–6–8–9–10–1–11

Path 4: 1–2–3–6–7–9–10–1–11

Note that each new path introduces a new edge. The path

1–2–3–4–5–10–1–2–3–6–8–9–10–1–11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges. Paths 1, 2, 3, and 4 constitute a ***basis set*** for the flow graph in Figure (d). That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity, V (G), for a flow graph, G, is defined as

V (G) = E – N + 2

Where *E* is the number of flow graph edges, *N* is the number of flow graph nodes.

3. Cyclomatic complexity, *V*(*G*), for a flow graph, *G,* is also defined as

V (G) = P + 1

Where *P* is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure (d), the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. V (G) = 11 edges – 9 nodes + 2 = 4.

3. V (G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure (d) is 4.

More important, the value for V (G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure ***average,*** depicted in PDL in Figure (f), will be used as an example to illustrate each step in the test case design method. Note that ***average,*** although an extremely simple algorithm contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules. Referring to the PDL for average in Figure (f), a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure (g).

2. Determine the cyclomatic complexity of the resultant flow graph. The cyclomatic complexity, V (G), is determined by applying the algorithms. It should be noted that V (G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1. Referring to Figure (g),

V (G) = 6 regions

V (G) = 17 edges – 13 nodes + 2 = 6

V (G) = 5 predicate nodes + 1 = 6

3. Determine a basis set of linearly independent paths. The value of V (G) provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six paths:

Path 1: 1-2-10-11-13

Path 2: 1-2-10-12-13

Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

```
PROCEDURE average;

*  This procedure computes the average of 100 or fewer
   numbers that lie between bounding values; it also computes the
   sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
   minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
   increment total.input by 1;
   IF value[i] > = minimum AND value[i] < = maximum
      THEN increment total.valid by 1;
           sum = s sum + value[i]
      ELSE skip
   ENDIF
   increment i by 1;
ENDDO
IF total.valid > 0
   THEN average = sum / total.valid;
   ELSE average = -999;
ENDIF
END average
```

Fig (f) PDL for the procedure average

4. Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set are

Path 1 test case:

value (k) = valid input, where $k < i$ for $2 \leq i \leq 100$

value (i) = – 999 where $2 \leq i \leq 100$

Expected results: Correct average based on k values and proper totals.

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = – 999

Expected results: Average = – 999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

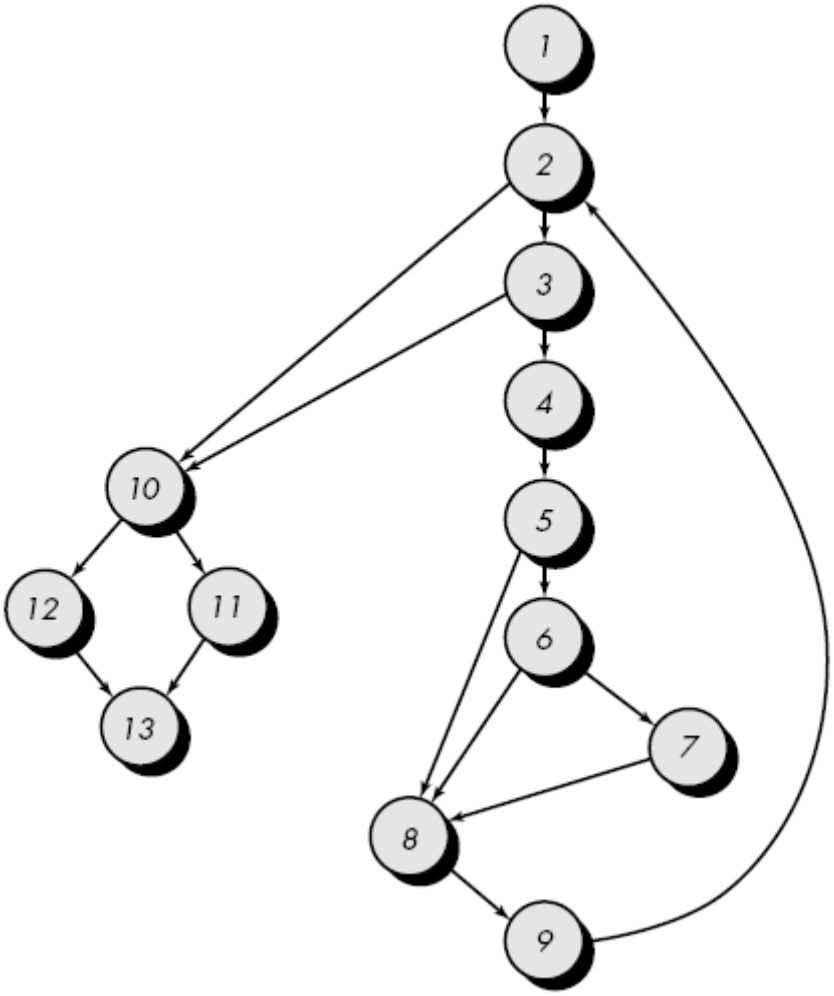


Fig (g) Flow graph for the procedure average

Path 4 test case:

value (*i*) = valid input where i < 100

value (*k*) < minimum where *k* < *i*

Expected results: Correct average based on k values and proper totals.

Path 5 test case:

value (*i*) = valid input where *i* < 100

value (*k*) > maximum where *k* <= *i*

Expected results: Correct average based on n values and proper totals.

Path 6 test case:

value (*i*) = valid input where *i* < 100

Expected results: Correct average based on n values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a ***graph matrix,*** can be quite useful. A ***graph matrix*** is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix is shown in Figure (h).

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b.*

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a ***link weight*** to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.
- The resources required during traversal of a link.

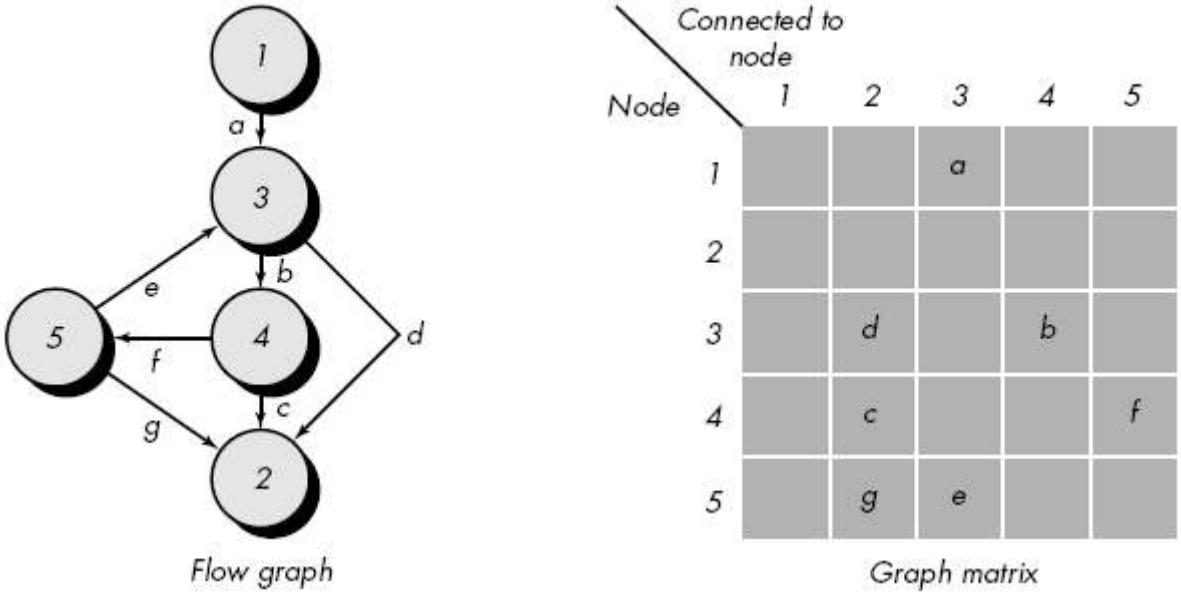


Fig (h) Graph Matrix

To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in Figure (h) is redrawn as shown in Figure (i). Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity). Represented in this form, the graph matrix is called a ***connection matrix.***

Referring to Figure (i), each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity.

Beizer provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

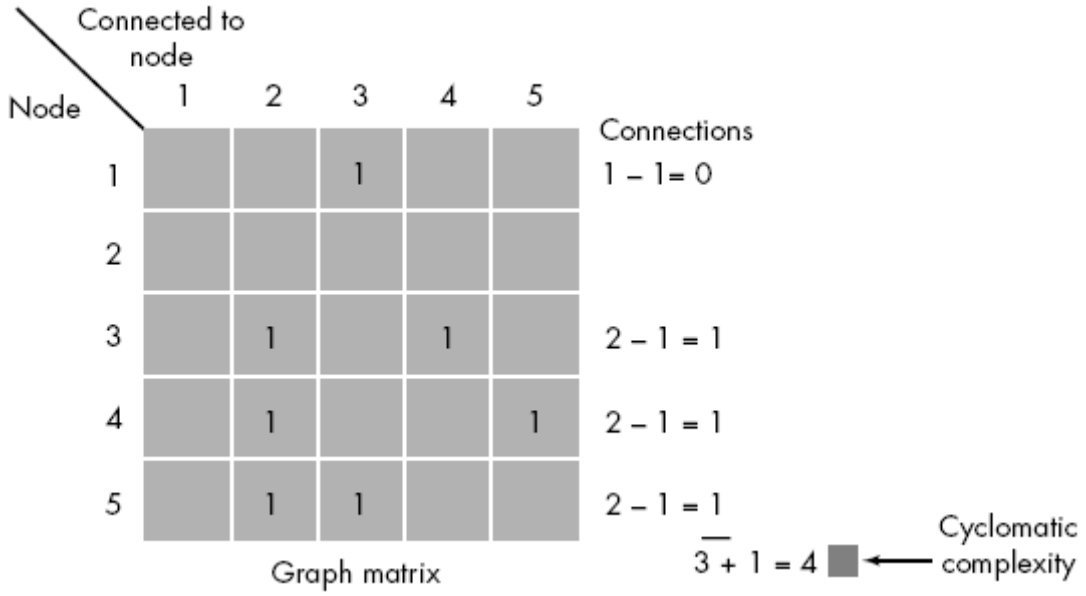
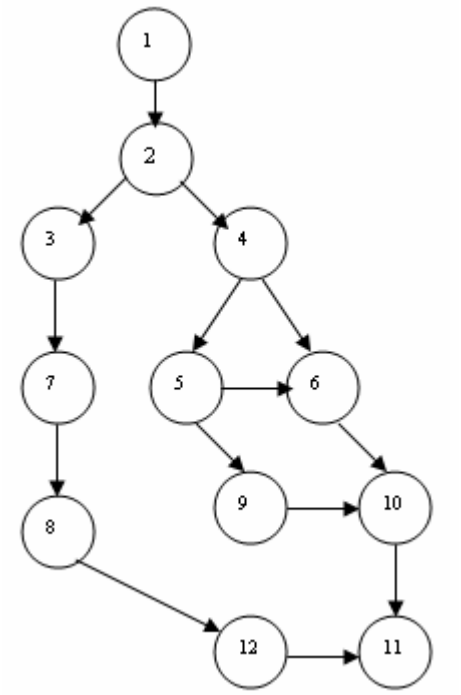


Fig (i) Connection Matrix

Q.1) Compute the cyclomatic complexity of a flow graph shown below and also find all independent paths:



Sol:

(a) Cyclomatic Complexity V (G) = E – N + 2

Here, E = 14

N = 12

So, V (G) = 14 – 12 + 2

V (G) = 4

(b) Cyclomatic Complexity V (G) = P + 1

Here, P = 3

So, V (G) = 3 +1

V (G) = 4.

(c) Cyclomatic Complexity V (G) = No of regions

So, V (G) = 4

(d) Independent Paths:

Path 1: 1 – 2 – 3 – 7 – 8 – 12 – 11

Path 2: 1 – 2 – 4 – 5 – 9 – 10 – 11

Path 3: 1 – 2 – 4 – 5 – 6 – 10 – 11

Path 4: 1 – 2 – 4 – 5 – 10 – 11

Q. 2) Determine the cyclomatic complexity for the function search.

```
void search (int a[ ], int n, int x)
{
        int i = 0, flag = 0;
        while (i<n)
        {
                if (a[i] == x)
                {
                        flag = 1;
                        break;
                }
                i = i +1;
        }
        if (flag == 1)
                printf("found");
        else
                printf("Not");
}
```

Sol:

The flow chart & flow graph of function search is shown below:

Cyclomatic Complexity V (G) = No. of Regions

V (G) = 4

Cyclomatic Complexity V (G) = E – N + 2

Here, E = 12

N = 10

So, V (G) = 12 – 10 + 2

V (G) = 4

Cyclomatic Complexity V (G) = P + 1

Here, P = 3 (Nodes 3, 4 and 7 are predicate nodes)

So, V (G) = 3 + 1

V (G) = 4

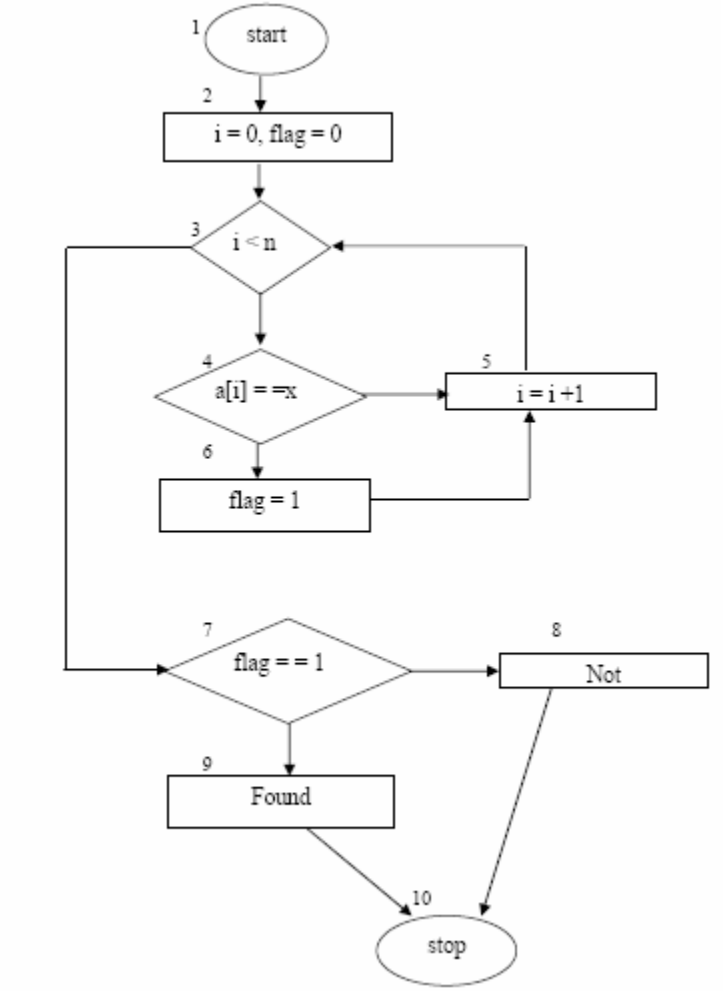


Fig Flow Chart

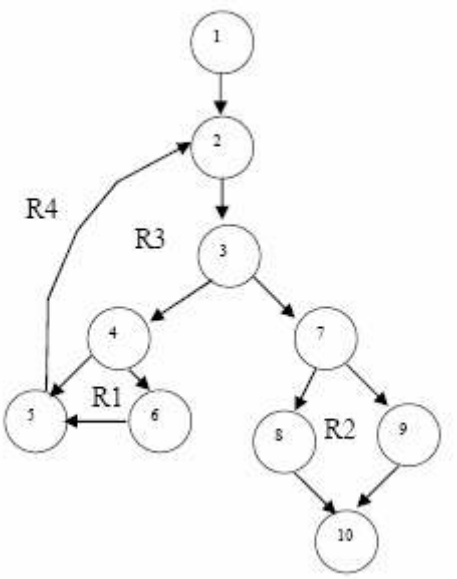


Fig Flow Graph

Q. 3) Write a function to check whether a given triangle is scalene, isosceles, equilateral, not a triangle for which lengths of three sides are given to you. Draw its flow graph. Find its cyclomatic complexity.

Sol: The Function for finding the triangle is scalene, isosceles, equilateral, not a triangle is as follows:

```
void check (int a, int b, int c)
{
        if (a==b && b==c && c==a)
                printf("Triangle is equilateral");
        else if (a!=b && b!=c && c!=a)
                printf("Triangle is scalene");
        else if (a==b || b==c || c==a)
                printf("Triangle is isosceles");
        else if ((a+b)<c || (b+c)<a || (a+c)<b)
                printf("Not a triangle");
}
```

The flow chart and flow graph of this function is shown below:

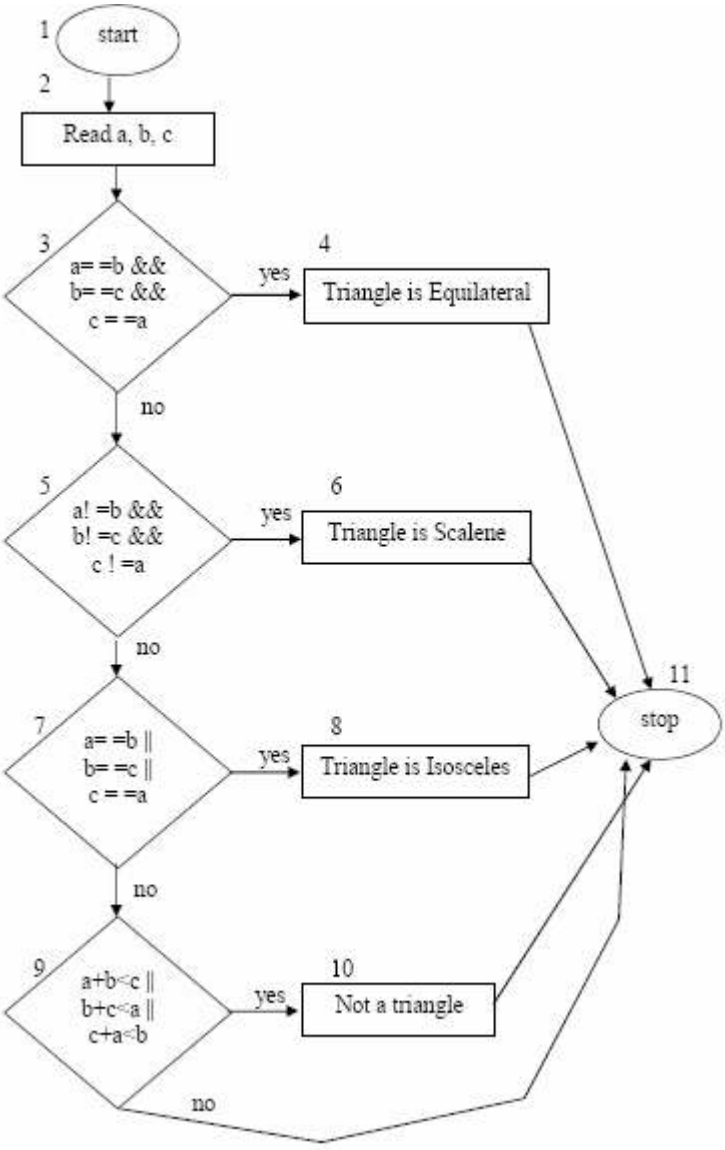


Fig Flow Chart

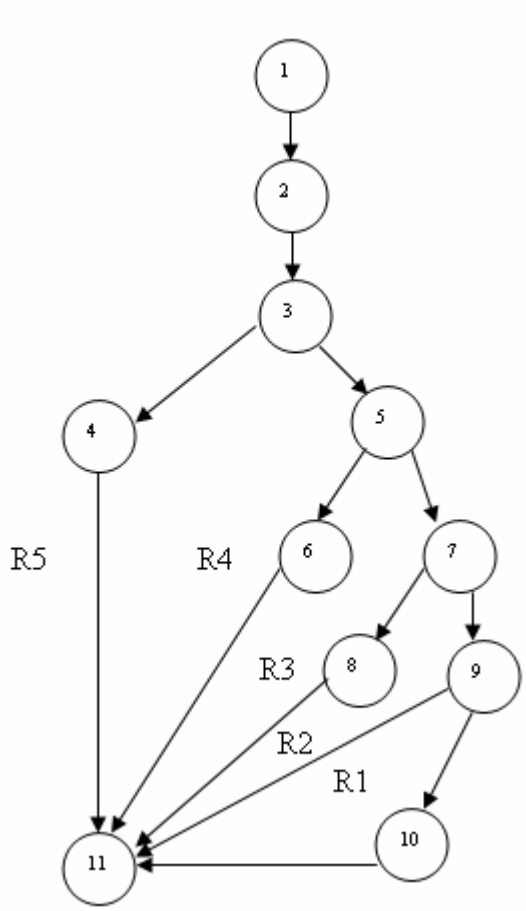


Fig Flow Graph

Cyclomatic Complexity V (G) = No. of regions

V (G) = 5

Cyclomatic Complexity V (G) = P + 1

Here, P = 4 (Nodes 3, 5, 7 and 9 are predicate nodes)

So, V (G) = 4 + 1

V (G) = 5

Cyclomatic Complexity V (G) = E – N + 2

Here E = 14, N = 11

So, V (G) = 14 – 11 + 2

V (G) = 5

The independent paths are:

Path 1: 1 – 2 – 3 – 4 – 11

Path 2: 1 – 2 – 3 – 5 – 6 – 11

Path 3: 1 – 2 – 3 – 5 – 7 – 8 – 11

Path 4: 1 – 2 – 3 – 5 – 7 – 9 – 11

Path 5: 1 – 2 – 3 – 5 – 7 – 9 – 10 – 11

Test cases are:

Path 1: test case:

a, b, c = valid input

Expected results: If a = = b && b = =c && c = = a then message “Triangle is equilateral” is printed.

Path 2: test case:

a, b, c = valid input

Expected results: If a != b && b !=c && c != a then message “Triangle is Scalene” is printed.

Path 3: test case:

a, b, c = valid input

Expected results: If a = = b || b = =c || c = = a then message “Triangle is isosceles” is printed.

Path 4: test case:

a, b, c = valid input

Expected results: If (a+b)<c || (b+c)<a || (c+a)<b then message “Not a triangle” is printed.

Path 5: test case:

a, b, c = valid input

Expected results: If all the conditions are failed then execution of function is stopped.

Q. 4) Write a function to find the different roots of a quadratic equation:

(i) Give flow graph.

(ii) Find Cyclomatic Complexity.

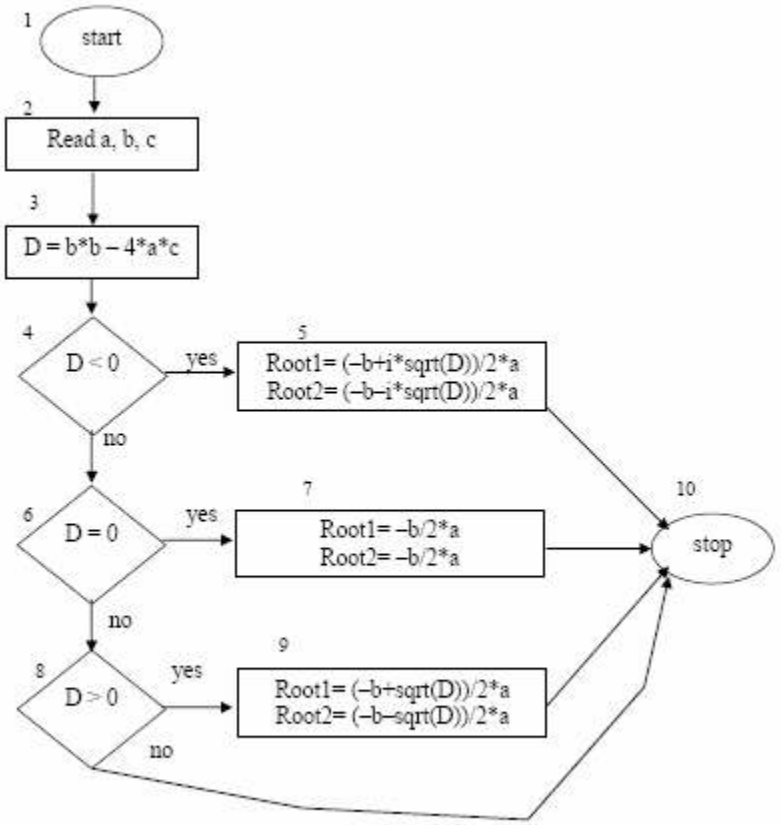
(iii) Find basic test paths.

(iv) Derive test cases.

Sol: The Function for finding the different roots for a quadratic equation is as follows:

```
void calc (int a, int b, int c)
{
        D = b*b – 4*a*c;
        if (D<0)
                Root1 = (–b+i*sqrt(D))/2*a;
                Root2 = (–b–i*sqrt(D))/2*a;
        else if (D ==0)
                Root1 = –b/2*a;
                Root2 = –b/2*a;
        else if (D>0)
                Root1 = (–b+sqrt(D))/2*a;
                Root2 = (–b+sqrt(D))/2*a;
}
```

The flow chart and flow graph of this function is shown below:



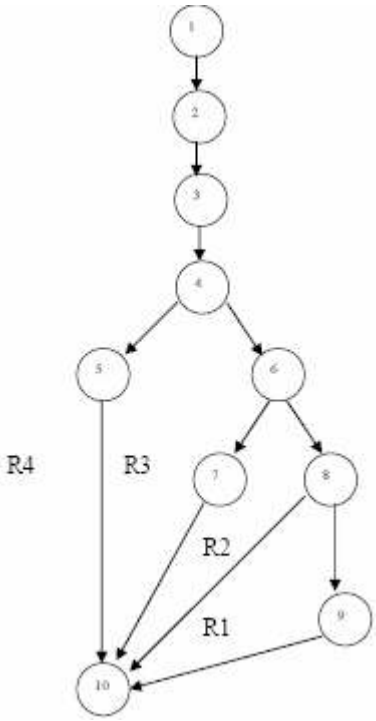


Fig Flow Chart Fig Flow Graph

Cyclomatic Complexity V (G) = No. of regions

V (G) = 4

Cyclomatic Complexity V (G) = P + 1

Here, P = 3 (Nodes 4, 6, and 8 are predicate nodes)

So, V (G) = 3 + 1

V (G) = 4

Cyclomatic Complexity V (G) = E – N + 2

Here E = 13

N = 11

So, V (G) = 13 – 11 + 2

V (G) = 4

The independent paths are:

Path 1: 1 – 2 – 3 – 4 – 5 – 11

Path 2: 1 – 2 – 3 – 4 – 6 – 7 – 11

Path 3: 1 – 2 – 3 – 4 – 6 – 8 – 9 – 11

Path 4: 1 – 2 – 3 – 4 – 6 – 8 – 10 – 11

Test cases are:

Path 1: test case:

a, b, c = valid input

Expected results: If D < 0 then Root1= (–b+i*sqrt(D))/2*a and Root2= (–b–i*sqrt(D))/2*a are printed.

Path 2: test case:

a, b, c = valid input

Expected results: If D = 0 then Root1= –b/2*a and Root2= –b/2*a are printed.

Path 3: test case:

a, b, c = valid input

Expected results: If D > 0 then Root1= (–b+sqrt(D))/2*a and Root2= (–b–sqrt(D))/2*a are printed.

Path 4: test case:

a, b, c = valid input

Expected results: If all the conditions are failed then execution of function is stopped.

Condition Testing

1. Condition testing is a test case design method that exercises the logical conditions contained in a program module.
2. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

   E1 <relational-operator> E2

   Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.
3. A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.
4. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.
5. If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:
   - Boolean operator error (incorrect/missing/extra Boolean operators).
   - Boolean variable error.
   - Boolean parenthesis error.
   - Relational operator error.
   - Arithmetic expression error.
6. The condition testing method focuses on testing each condition in the program.
7. Condition testing strategies generally have two advantages.
   a) Measurement of test coverage of a condition is simple.
   b) The test coverage of conditions in a program provides guidance for the generation of additional tests for the program.
8. The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program *P* is effective for detecting errors in the conditions contained in *P,* it is likely that this test set is also effective for detecting other errors in *P.* In addition, if a testing strategy is effective for detecting errors in a condition, then it is likely that this strategy will also be effective for detecting errors in a program.

Data Flow Testing

1. The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.
2. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

DEF(S) = {X | statement S contains a definition of X}

USE(S) = {X | statement S contains a use of X}

If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

3. A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.
4. One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy.
5. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the then part has no definition of any variable and the else part does not exist. In this situation, the else branch of the if statement is not necessarily covered by DU testing.
6. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.
7. To illustrate this, consider the application of DU testing to select test paths for the PDL that follows:

```
proc x
        B1;
        do while C1
                if C2
                        then
                                if C4
                                        then B4;
                                        else B5;
                                endif;
                        else
                                if C3
                                        then B2;
                                        else B3;
                                endif;
                        endif;
                enddo;
                B6;
        end proc;
```

8. To apply the DU testing strategy to select test paths of the control flow diagram, we need to know the definitions and uses of variables in each condition or block in the PDL. Assume that variable X is defined in the last statement of blocks B1, B2, B3, B4, and B5 and is used in the first statement of blocks B2, B3, B4, B5, and B6. The DU testing strategy requires an execution of the shortest path from each of Bi, $0 < i \leq 5$, to each of Bj, $1 < j \leq 6$. (Such testing also covers any use of variable X in conditions C1, C2, C3, and C4.) Although there are 25 DU chains of variable X, we need only five paths to cover these DU chains. The reason is that five paths are needed to cover the DU chain of X from Bi, $0 < i \leq 5$, to B6 and other DU chains can be covered by making these five paths contain iterations of the loop.
9. If we apply the branch testing strategy to select test paths of the PDL just noted, we do not need any additional information. To select paths of the diagram for BRO testing, we need to know the structure of each condition or block. (After the selection of a path

of a program, we need to determine whether the path is feasible for the program; that is, whether at least one input exists that exercises the path.) Since the statements in a program are related to each other according to the definitions and uses of variables, the data flow testing approach is effective for error detection. However, the problems of measuring test coverage and selecting test paths for data flow testing are more difficult than the corresponding problems for condition testing.

Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

Simple loops – The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m < n.
5. n – 1, n, n + 1 passes through the loop.

Nested loops – If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated loops – Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops – Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

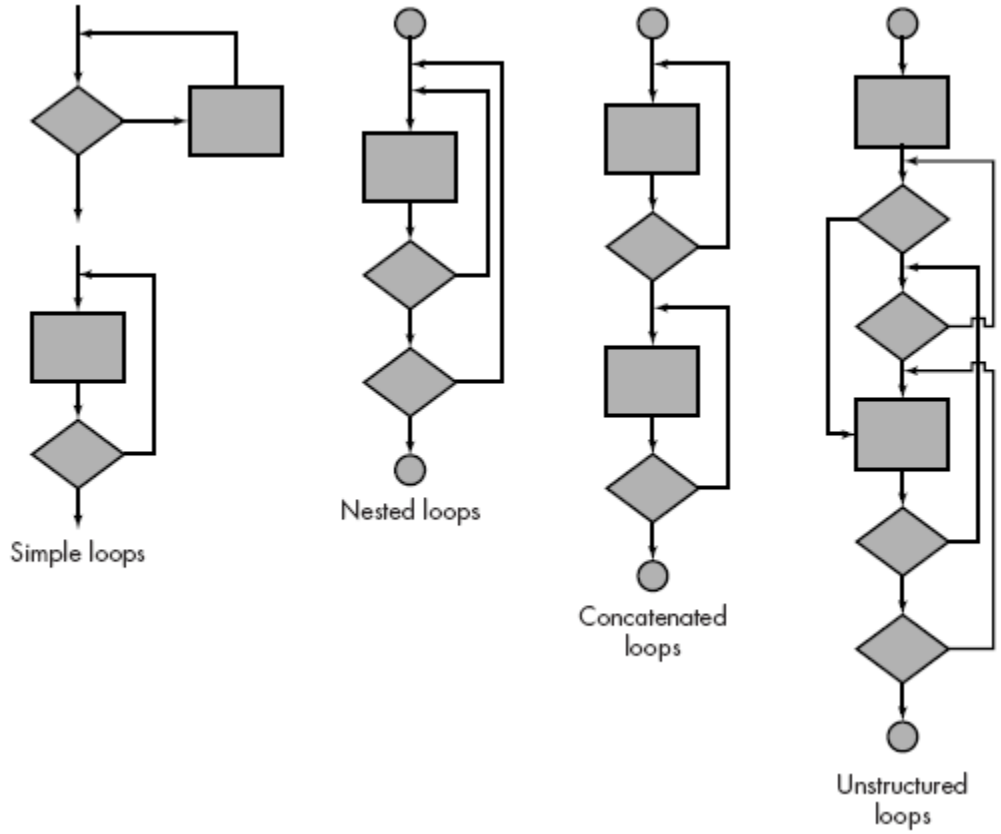


Fig Classes of Loops

Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either, a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence classes are defined.

4. If an Input condition is *Boolean,* one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application. The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form

area code—blank or three-digit number

prefix—three-digit number not beginning with 0 or 1

suffix—four-digit number

password—six digit alphanumeric string

commands—check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking application can be specified as

area code: Input condition, *Boolean*—the area code may or may not be present.
Input condition, *range*—values defined between 200 and 999, with specific exceptions.

prefix: Input condition, *range*—specified value >200
Input condition, value—four-digit length

password: Input condition, *Boolean*—a password may or may not be present.
Input condition, *value*—six-character string.

command: Input condition, *set*—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest numbers of attributes of an equivalence class are exercised at once.

Boundary Value Analysis

For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that ***boundary value analysis*** (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

A Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure below. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

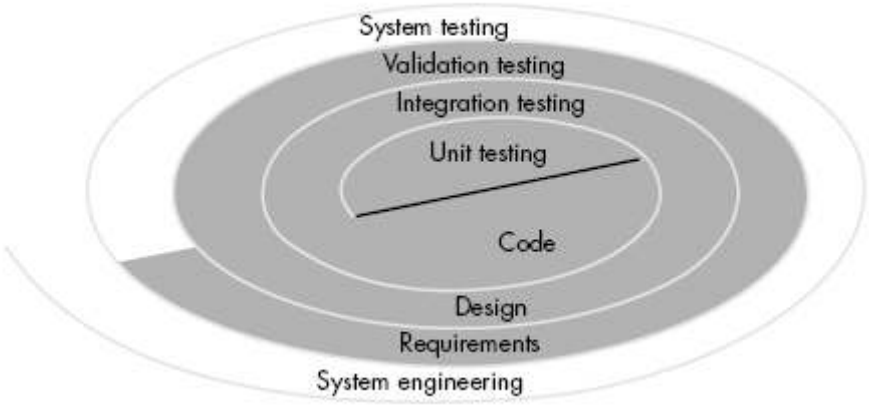


Fig A testing Strategy

A strategy for software testing may also be viewed in the context of the spiral. ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to ***integration testing,*** where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter ***validation testing,*** where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at ***system testing,*** where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.

Various Levels of Testing

If we focus the process of testing in context of software engineering, it is actually a series of four steps that are implemented sequentially. Each step known as level is meant for test different aspects of the system. The basic levels of testing are

- Unit Testing
- Validation Testing
- Integration Testing
- System Testing

The relation of the fault introduced indifferent phase and different levels of testing are shown below:



1. **Unit Testing** – This testing is essentially for verification of the code produced during the code phase. The basic objective of this phase is to test the internal logic of the module. Unit testing makes heavy use of the white box testing technique, exercising specific path in a module control structure to ensure complete coverage and maximum error detection. In Unit testing the module interface is tested to ensure that information properly flows into and out of the program unit. Also the data structure is tested to ensure that data stored temporarily maintain its integrity during execution or not.
2. **Integration Testing** – This testing uses the address of verification and program construction. Black box testing technique is widely used in this testing strategy although limited amount of white box testing may be used to ensure coverage of control paths. The basic emphasis of this testing the interfaces between the modules.

3. **Validation Testing** – Criterion, which is established during requirement analysis, is tested and it provides final assurance that software meets all functional behavior, performance requirement etc.
4. **System Testing** – Here the entire software is tested. The last high order testing stage falls outside the boundary of software engineering. System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

Guidelines for a Successful Testing Strategy

1. Specify product requirements in a quantifiable manner long before testing commences.
2. State testing objectives explicitly.
3. Understand the users of the software and develop a profile for each user category.
4. Develop a testing plan that emphasizes "rapid cycle testing."
5. Build "robust" software that is designed to test itself.
6. Use effective formal technical reviews as a filter prior to testing.
7. Conduct formal technical reviews to assess the test strategy and test cases themselves.
8. Develop a continuous improvement approach for the testing process.

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure below. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

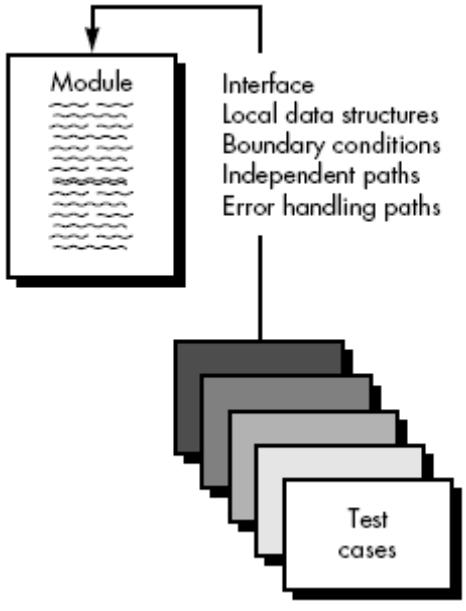


Fig Unit Test

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Among the more common errors in computation are

(1) Misunderstood or incorrect arithmetic precedence,

(2) Mixed mode operations,

(3) Incorrect initialization,

(4) Precision inaccuracy,

(5) Incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as

(1) Comparison of different data types,

(2) Incorrect logical operators or precedence,

(3) Expectation of equality when precision error makes equality unlikely,

(4) Incorrect comparison of variables,

(5) Improper or nonexistent loop termination,

(6) Failure to exit when divergent iteration is encountered, and

(7) Improperly modified loop variables.

Among the potential errors that should be tested when error handling is evaluated are

1. Error description is unintelligible.
2. Error noted does not correspond to error encountered.
3. Error condition causes system intervention prior to error handling.
4. Exception-condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the last (and probably most important) task of the unit test step. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

Unit Test Procedures

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

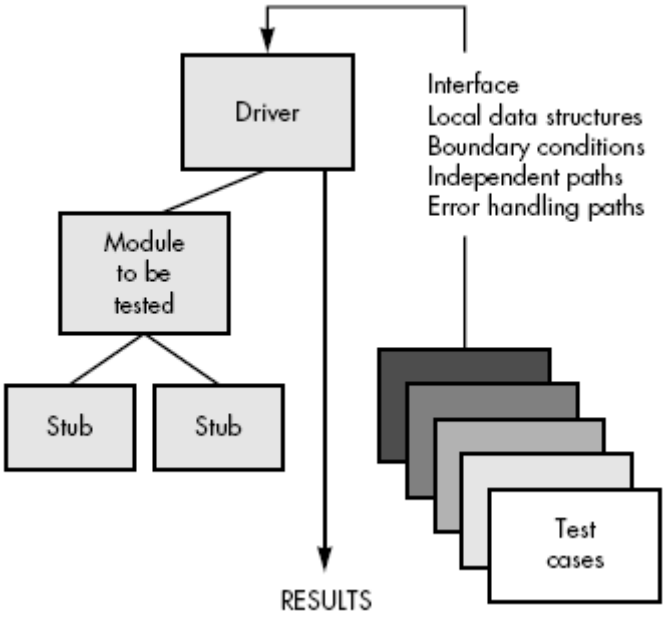


Fig Unit Test Environment

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure above. In most applications a ***driver*** is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. ***Stubs*** serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing. Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used). Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Advantage of Unit Testing

1. Can be applied directly to object code and does not require processing source code.
2. Performance profilers commonly implement this measure.

Disadvantages of Unit Testing

1. Insensitive to some control structures (number of iterations)
2. Does not report whether loops reach their termination condition
3. Statement coverage is completely insensitive to the logical operators (|| and &&).

Integration Testing

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design. There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-down Integration

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

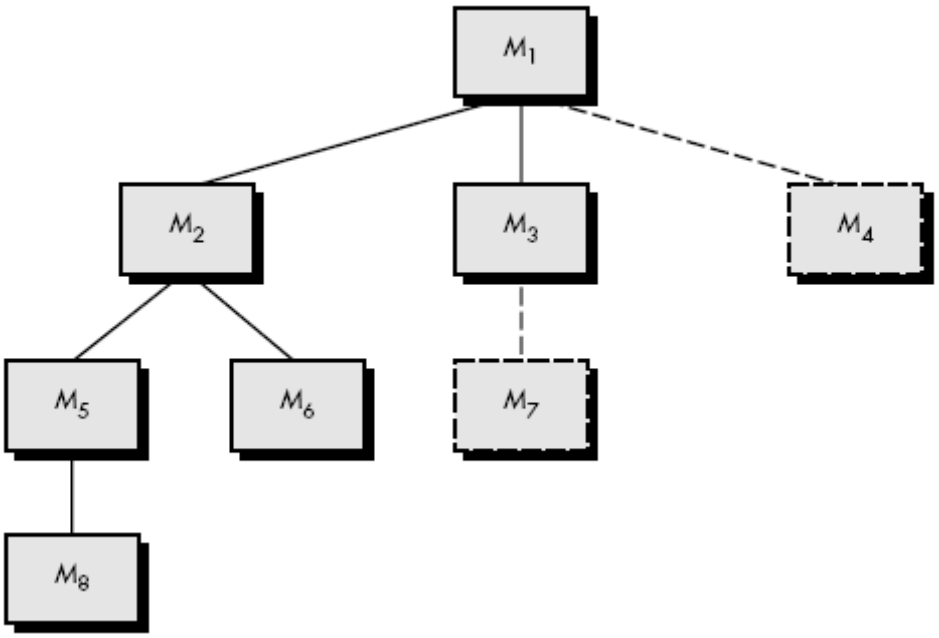


Fig Top – Down Integration

Referring to Figure above, ***depth-first integration*** would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. ***Breadth-first integration*** incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path. The incoming path may be integrated in a top-down manner. All input processing (for subsequent transaction dispatching) may be demonstrated before other elements of the structure have been integrated. Early demonstration of functional capability is a confidence builder for both the developer and the customer.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. The tester is left with three choices:

(1) Delay many tests until stubs are replaced with actual modules,

(2) Develop stubs that perform limited functions that simulate the actual module, or

(3) Integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) causes us to loose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex.

Bottom-up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with ***atomic modules*** (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called ***builds***) that perform a specific software sub-function.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

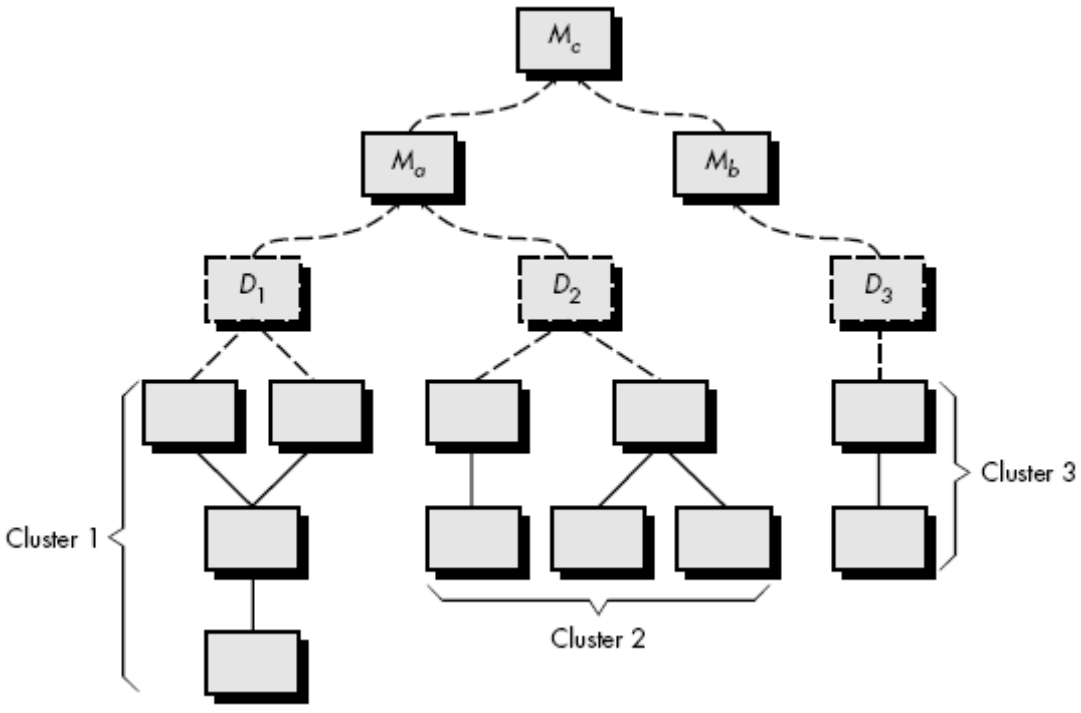


Fig Bottom up Integration

Integration follows the pattern illustrated in Figure above. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed and

the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, ***regression testing*** is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.* Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Smoke Testing

Smoke testing is an integration testing approach that is commonly used when "shrink – wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell describes the smoke test in the following manner:

"The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly".

Smoke testing provides a number of benefits when it is applied on complex, time critical software engineering projects:

- ***Integration risk is minimized***. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- ***The quality of the end-product is improved***. Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.
- ***Error diagnosis and correction are simplified***. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- ***Progress is easier to assess***. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

Validation Testing

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests—***validation testing***—may begin. Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the ***Software Requirements Specification***— a document that describes all user-visible attributes of the software. The specification contains a section called ***Validation Criteria***. Information contained in that section forms the basis for a validation testing approach.

Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exists:

(1) The function or performance characteristics conform to specification and are accepted or

(2) A deviation from specification is uncovered and a ***deficiency list*** is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

Configuration Review

An important element of the validation process is a ***configuration review***. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle.

Alpha Testing

The ***alpha test*** is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

Beta Testing

The ***beta test*** is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

System Testing

Software is the only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and

(1) Design error-handling paths that test all information coming from other elements of the system,

(2) Conduct a series of tests that simulate bad data or other potential errors at the software interface,

(3) Record the results of tests to use as "evidence" if finger-pointing does occur, and

(4) Participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

Recovery Testing

Many computer based systems must recover from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check-pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack." During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

(1) Special tests may be designed that generate ten interrupts per second, when one or two is the average rate,

(2) Input data rates may be increased by an order of magnitude to determine how input functions will respond,

(3) Test cases that require maximum memory or other resources are executed,

(4) Test cases that may cause thrashing in a virtual operating system are designed,

(5) Test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called ***sensitivity testing***. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. ***Performance testing*** is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

Debugging

Software testing is a process that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

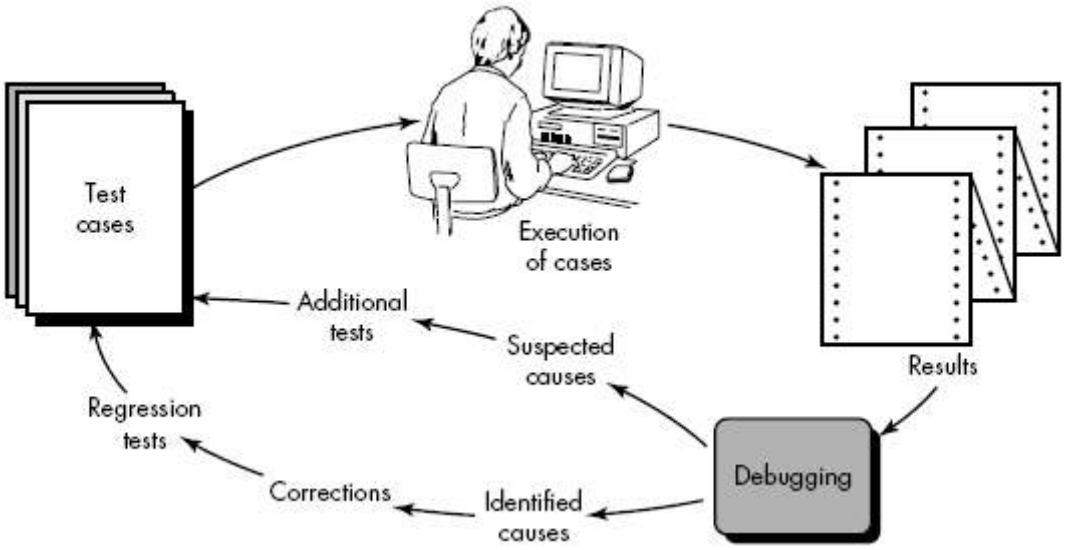


Fig Debugging Process

The Debugging Process

Debugging is not testing but always occurs as a consequence of testing. Referring to Figure above, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes:

(1) The cause will be found and corrected, or

(2) The cause will not be found.

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g. the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches

In general, three categories for debugging approaches may be proposed:

(1) Brute force,

(2) Backtracking, and

(3) Cause elimination.

1. The ***brute force*** category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

2. ***Backtracking*** is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.
3. The third approach to debugging – ***Cause elimination***—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Software Quality Assurance (SQA)

Quality

It's a measurable quantity which can be compared with known standards for any product or process. It specifies the degree to which a product or process maintains the known standards. Quality can be classified in to two groups – a) Quality of design and b) Quality of conformance.

a) **Quality of design:** - It refers to the standards maintained during design processes which include the grade of - i) Materials, ii) Tolerances and iii) Performance specifications.
b) **Quality of conformance**: - It refers to the degree to which the design specifications are followed during manufacturing.

Quality Control

a) It's a series of inspections, reviews and tests used throughout the development cycle to ensure that each work product meets the requirements placed up on it.
b) It includes a feedback loop to the process that created the work product. The feedback loop helps tune up the process during the creation of the product.
c) It should be a part of the manufacturing process.

Quality Assurance

It's the auditing and reporting functions of the management. If the data provided through quality assurance identify problems then it's the management's responsibility to solve the problems and apply the necessary resources to resolve the quality issues. The goal of quality assurance is to provide management to provide information and data related to product quality so that they can evaluate whether or not the manufacturing process is maintaining the product quality.

Costs related to maintenance of Quality

The Quality related costs are divided in to following categories: -

a) Prevention (from degraded quality) costs,
b) Appraisal (inspection) costs and
c) Failure (to maintain quality) costs.

a) **Prevention costs** :- It includes costs required to implement the following –
   i) Quality planning,
   ii) Formal technical reviews,
   iii) Test equipment and
   iv) Training.

b) **Appraisal costs**: - It includes activities to gain the quality standard of a product when going through the process for the first time. Examples include –

i) In-process and inter process inspections,

ii) Testing,

iii) Equipment calibration and maintenance.

c) **Failure costs**: - These costs are raised when a failure occurs other wise these costs get avoided. Failure costs can be further divided in to following –

I. **Internal failure costs**: - These are incurred when the defects are detected before the product gets delivered to the customer. The activities included here are –

1) Rework,

2) Repairs and

3) Failure mode analysis.

II. **External failures**: - These are the costs incurred when the defects are detected after the product has reached the user. Examples are –

1) Complaint resolution,

2) Product return and replacement,

3) Warranty support, etc.

Software Quality Assurance (SQA) Activities

The activities performed in software quality assurance involves two groups –

a) The software engineers doing the technical work: - Their work includes –

1) Applying technical methods and measures for ensuring high quality,

2) Conducting formal reviews and

3) Performing well planned software testing.

b) The SQA group :- That has the responsibility for –

1) Quality assurance planning,

2) Oversight,

3) To assist the software engineering team in achieving high standards of production,

4) Record keeping,

5) Analysis and

6) Reporting.

The SQA activities performed by both SQA groups and Software engineering group are governed by an SQA plan for the project. An SQA plan consists of the following activities:-

1) **The Management section**: - It consists of the following standards –

a) The organizational structure for the project,

b) Tasks and activities to be maintained in the organizational structure,

c) The organizational roles and responsibilities for maintaining good product quality.

2) **The Documentation section**: - It describes the standards to be maintained in the following elements –

a) Project documents (e.g. project plan),

b) Models (e.g. class hierarchies),

c) Technical documents (e.g. test plans, specifications),

d) User documents (e.g. help files).

3) **Standards, Practices and Conventions section**: - It contains the list of applicable standards and practices to be followed during the software process. It requires that all project, process and product matrices that are used in the project should be listed out.
4) **Reviews and Audits section**: - It describes the quality reviews and quality audits that are to be conducted by the software engineering group, the SQA group and the customer.
5) **The Test section**:-It refers to the software test plan and testing procedures. It also refers to the standards to be maintained for keeping the records of the tests performed.
6) **Problem reporting and corrective action section**: - It defines the procedures for reporting, tracking and resolving errors & defects. It also specifies the organizational responsibilities for these activities.

Statistical Quality Assurance

It implies the following steps –

1) Information about software defects is collected and categorized.
2) An attempt is made to trace the cause of each defect to its basis.
3) Using the Pareto principle to find out the vital view which states that 80 percent of the defects can be traced to 20 percent of all possible causes. Which means that if all the errors are found and all the possible causes for those errors are also found, then its observed that about 20 percent of the possible causes would justify about 80 percent of the total errors. This 20 percent is collectively called the "vital few".
4) Once the vital view causes are identified then the procedure of error correction should be carried out.

Software Maintenance

Software Maintenance is a very broad activity that includes error, corrections, enhancement of capabilities, deletion of obsolete capabilities and optimization. Because change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications. So any work done to change the software after it is in operation is considered to be maintenance. The purpose is to preserve the value of software overtime. The value can be enhanced by expanding the customer base, meeting additional requirements, becoming easier to use, more efficient and employing newer technology. Maintenance may span for 500 years, whereas development may be 1–2 years.

Types of Maintenance

There are four major categories of maintenance

1. **Corrective Maintenance** – This refers to modifications initiated by defects in the software. A defect can result from, **design errors, logic errors** and **coding errors**. **Design errors** occur when, changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. **Logic errors** result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow or incomplete test data. **Coding errors** are caused by incorrect implementations of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors.

   In the event of system failure due to an error, actions are taken to restore operation of the software system. Due to pressure from management, maintenance personnel sometimes resort to emergency fixes known as patching. The adhoc nature of this approach often gives rise to a range of problems that include increased program complexity and unforeseen ripple effects. Unforseen ripple effects imply that a change to one part of a program may affect other sections in an unpredictable manner, thereby leading to distortion in the logic of the system. This is often due to lack of time to carry out a through "impact analysis" before effecting the change.

2. **Adaptive Maintenance** – It includes modifying the software to match changes in the over changing environment. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the software. For example, business rules, government policies, work patterns, software and hardware operating platforms. A change to the whole or part of this environment will require corresponding modifications of the software.
   Thus, this type of maintenance includes any work initiated as a consequence of moving the software to a different hardware or software platform – compiler, operating system or new processor. Any change in the government policy can have far reaching ramification on the software. When European countries had decided to for "single European currency", this change affected all banking system software and was modified accordingly.
3. **Perfective Maintenance** – It means improving processing efficiency or performance, or restructuring the software to improve changeability. When the software becomes useful, the user tends to experiment with new cases beyond the scope for which it was initially developed. Explosion in requirements can take form of enhancement of existing system functionality or improvement in computational efficiency; for example, providing a Management Information System with a data entry module or a new message handling facility.
4. **Preventive Maintenance** – There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflects deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance. This term is often used with hardware systems and implies such things as lubrication of parts before need occurs, or automatic replacement of banks of light bulbs before they start to individually burn out.

Unit 5

Software Project Management

Introduction

Building computer software is a complex undertaking task, which particularly involves many people working over a relatively long time. That's why software projects needs to be managed. The software project management is the first layer of software engineering process. It starts before the technical work starts, continues as the software evolves from conceptual stage to implementation stage. It is a crucial activity because the success and failure of the software is directly depends on it.

Software project management is needed because professional software engineering is always subject to budget constraints, schedule constraints and quality oriented focus.

Definition

Project management involves the planning, monitoring and control of the people, process and events that occurs as software evolves from a preliminary concept to an operational implementation. Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development and support of computer software. The project management activity encompasses measurements and metrics estimation, risk analysis, schedules, tracking and control.

Effective software project management focuses on the **four P's**: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the product.

1. The People

a) The "people factor" is so important that the Software Engineering Institute has developed a ***people management capability maturity model*** (**PM-CMM**).

b) To enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability.

c) The PM-CMM is a companion to the software capability maturity model that guides organizations in the creation of a mature software process.

d) The PMCMM defines the following areas for software people.

   a. Recruiting
   b. Selection
   c. Performance Management
   d. Training
   e. Career Development
   f. Team Culture Development

2. The Product

Before a project can be planned,

a) Product objectives and scope should be established.

b) Alternative solutions should be considered.

c) Technical and management constraints should be identified.

d) The software developer and customer must meet to define product objectives and scope.

e) Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved.

f) Scope identifies the primary data, functions and behaviors that characterize the product, and more important, attempts to *bound* these characteristics in a quantitative manner.

3. The Process

a) A software process provides the framework from which a comprehensive plan for software development can be established.

b) A small number of framework activities are applicable to all software projects, regardless of their size or complexity.

c) A number of different tasks set—tasks, milestones, work products and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.

d) Umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model.

4. The Project

a) We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity.

b) The overall development cycle is called as Project.

c) In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring and controlling the project.

The Software Team

The best team structure for any particular project depends on the nature of the project & product and also the individual characteristics of the team members. The basic team structures are as follows – a) Democratic teams, b) Chief programmer teams, c) Hierarchical team.

1. Democratic Teams or Democratic decentralized (DD)

It has following features.

1) The team leader position does not rotate among the team members because a team functions best when one individual is responsible for coordinating team activities and for making final decisions in situations where collective decisions can not work.
2) Here all the decisions are made by collective effort of the members.
3) All the activities carried out during project are collectively discussed and handled.

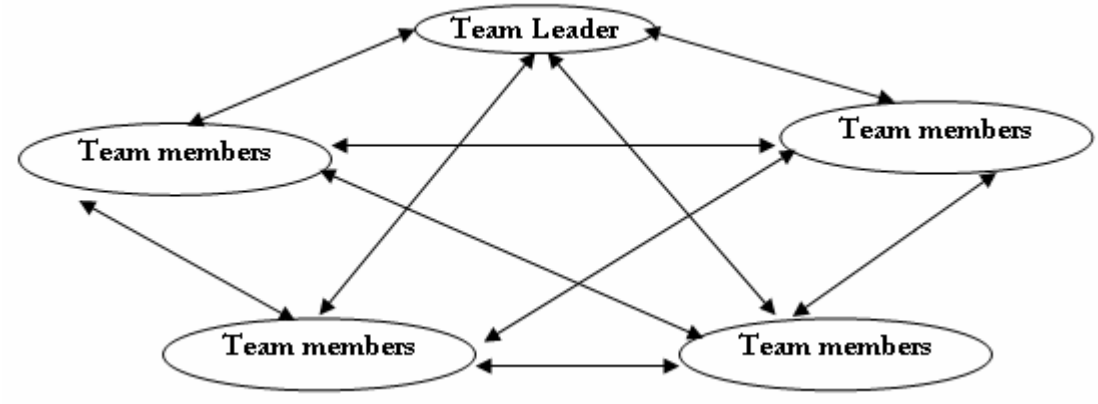


Fig Democratic Team Structure

Advantages: -

I) Opportunity for team members to contribute to decisions.

II) Opportunity for team members to learn from each other.

III) Increased job satisfaction due to equal importance and non threatening environment.

IV) These teams can stay together for several years and may work on several different projects.

Disadvantages:-

I) Communications overhead required for reaching to collective decisions.

II) A lot of coordination required between team members.

III) Less individual responsibility and authority results in less personal drive and initiative from team members.

2. Chief Programmer Teams or Controlled decentralized (CD)

It has following features.

1) They are highly structured
2) The chief programmer designs the product and makes all the decisions.
3) The chief programmer implements the critical parts of the project.
4) The chief programmer allocates the work for the individual programmer under him.
5) Usually the number of programmers ranges from 2 to 5 only.
6) The programmers do the coding, debug; document and unit test the system.
7) The chief programmer is assisted by a backup consultant programmer on various technical problems, provides connection with the customer provides interaction with quality assurance group and may participate in analysis, design and implementation phases.
8) The chief programmer is also assisted by an administrative program manager, who handles the administrative details which includes time cards for the employees, sick leave and vacation schedule.

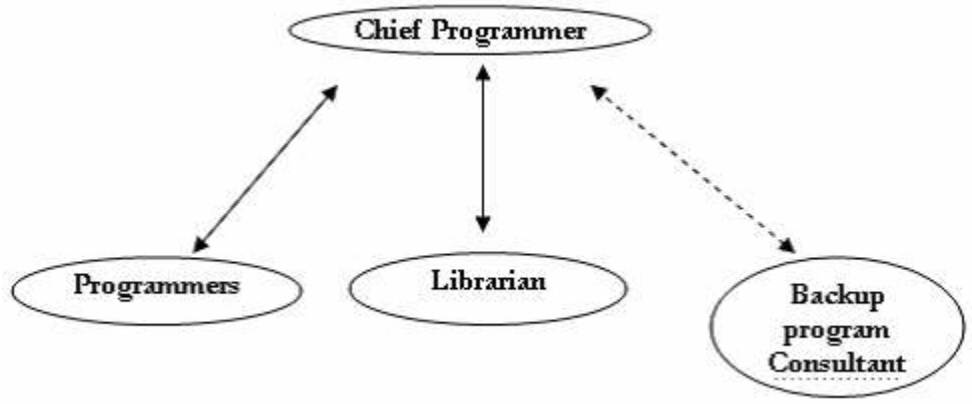


Fig Chief Programmer Team Structure

Advantages:-

I) Centralized decision making reduces the decision making time.

II) It reduces communication paths and related overheads.

Disadvantages:-

I) As all the decisions are taken by the chief programmer, hence it results in low moral among the programmers.

II) The effectiveness of this structure depends solely on the efficiency and knowledge of the chief programmer.

3. Hierarchical Team or Controlled Centralized (CC)

It has following features.

1) It s a mixed approach of Democratic and Chief programmer team structures.
2) Here the project leader has under his control, 2 to 5 senior programmers who individually have 5 to 7 junior programmers under their control.
3) The various jobs of the Project leader includes –

a) Assigning tasks,

b) Attending reviews and walkthroughs,

c) Detecting problem areas,

d) Balancing of the work load,

e) Participation in various technical activities.

4) The major decisions are taken by the Project leader and who in-turn gives some decision making power to the senior programmers also.

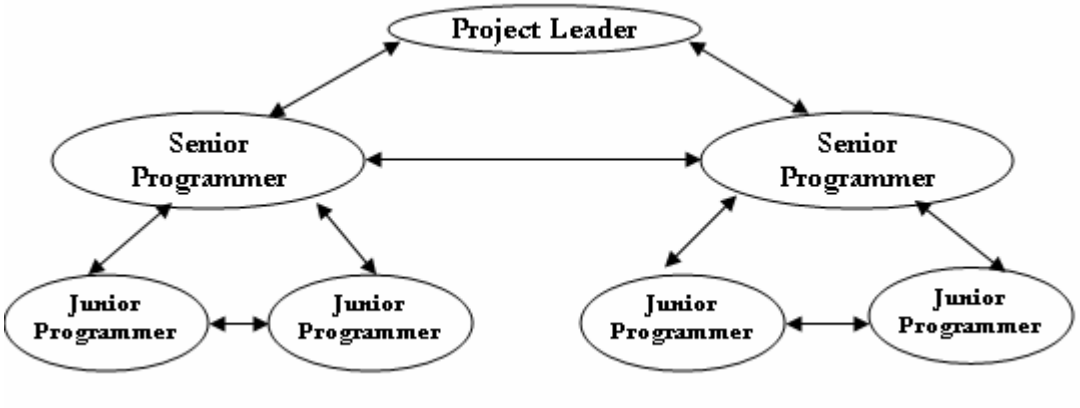


Fig Hierarchical Team Structure

Advantages:-

I) Here the number of communication paths are limited hence permitting effective communication

II) Here the time span required for deciding and implementing the decided processes takes less time

III) The job satisfaction is fairly good as the scope of promotions is good.

Disadvantages:-

I) The most technically efficient programmers tend to be promoted, so the best programmers are lost.

II) The best programmers may not be good managers hence promoted to a management post might result in reduction in productivity.

Software Project Planning – Following are the various project planning activities

1) Defining the problem –

a) Develop a definitive statement of the problem to be solved, including a description of the present situation, problem constraints and a statement of the goals to be achieved. The problem statement should be formulated in the customer's terminology.

b) Justify a computerized solution strategy for the problem.

c) Identify the various functions provided by the hardware subsystem, software subsystem and people subsystem. The constraints thereof also should be identified.

d) Determining system level goals and requirements for the development process and the work products.

e) Establish high-level acceptance criteria for the system.

2) Developing a solution strategy –

a) Create multiple solution strategies, with out considering the constraints.

b) Conduct a feasibility study for each strategy.

C) Recommend a solution strategy, indicating why other strategies were rejected.

d) Develop a list of the characteristics of the product priority wise.

3) Planning the development process –

a) Define a life-cycle model and an organizational structure for the project.

b) Planning of –

I) Configuration management activities,

II) Quality assurance activities and

III) Validation activities.

c) Determine phase-dependent factors –

I) Tools,

II) Techniques, and

III) Notations.

d) Establish preliminary cost estimates for the system development.

e) Establish a preliminary development schedule.

f) Establish preliminary staffing estimates.

g) Preliminary estimation of the required computing resources and maintenance of those systems.

h) Preparation of glossary of terms.

i) Identification of information sources for reference during the project development.

Detail Explanation of Software Project Planning activities

Problem Definition

There is a need to prepare a concise statement of the problem to be solved and the constraints that exist for its solution in customer's terminology. Problem definition requires understanding of the problem domain and the problem environment. Techniques for gaining this knowledge include –

a) Customer interviews,

b) Observation of the problem tasks, and

c) Actual performance of the tasks developed by the planner. Here the planner should not be biased in any way and should certainly be technically experienced.

After preparing the solutions the successive task includes the determination of the -

a) Appropriateness of the computerized solution,

b) Cost-effectiveness,

c) It should avoid displacing existing workers as it may not be acceptable in the society. etc.

Requirement analysis and Goal determination:-

Goals:

2) Goals are targets for achievement and serve to establish the frame work for a software development project.
3) Goals apply to both the development process and the work products.
4) Goals can be either qualitative or quantitative. They have the following categories-
    a) **Qualitative process goal :-** the development process should adhere to quality observed under quality assurance.
    b) **Quantitative process goal :-** the system should be delivered with in a fixed time.
    c) **Qualitative product goal :-** the system should make the user's job more easy & interesting.
    d) **Quantitative product goal :-** the system should reduce the cost of transaction by about 25 %.
5) Other common goals include a) transportability, b) early delivery, c) ease for nonprogrammers etc.

Requirement:

2) They include –
    a) Functional aspects,
    b) Performance aspects,
    c) Hardware aspects,
    d) Software aspects,
    e) User interface, etc.
3) They also specify development standards and quality assurance standards for both project process and product.
4) Special efforts should be made in the process of developing meaningful requirement statements and methods that will be used to verify those statements.

During goal determination and requirement analysis the quality attributes of the software has to be taken in to consideration. According to IEEE standards following are the desired quality attributes of a software product:-

1) Portability ,
2) Reliability,
3) Efficiency,
4) Accuracy,
5) Error,
6) Robustness,
7) Correctness.

Developing a Solution Strategy

1) A solution strategy is not a detailed solution plan. It's a general statement of solution concerning the nature of possible solutions.
2) A solution strategy should account for all external factors that are visible to the product users.
3) A strategy should be phrased to permit alternative approaches to product design.
4) Several strategies should be considered before one is adopted.
5) Solution strategies should be generated with out regard for feasibility because its not possible to be both creative & critical at the same time.
6) Often the best strategy is composite of ideas from several different approaches. And the best solution strategies may become apparent only after all the obvious solutions have been enumerated.
7) The feasibility of each proposed solution strategy can be established by examining solution constraints. Constraints prescribe the boundaries of the solution space.
8) A solution strategy is feasible if the project goals and requirements can be satisfied with in the constraints of available time , resources and technology using that strategy.
9) When recommending a solution strategy its extremely important to document the reasons for rejecting other strategies, this provides justification for the recommended strategy and may prevent ill-considered revisions at some later date.

A solution strategy should include a priority list of product features. There are several important reasons for stating product priorities. At some later time in the development cycle it may be necessary to postpone or eliminate some system capabilities due to inconsistencies in the requirements, technical bottlenecks or time and cost overruns.

Planning the Software Development Processes

A Software process consists of two parts

1) **Product Engineering Processes**, which consists of
   a) **Development process**: - This process consists of all kinds of activities which contribute towards the software development process which are performed by the programmers, designers, testing personnel, librarians etc. The major goal here is to improve quality of the software product.
   b) **Project Management process**: - This process includes all the activities related to the efficient management of various stages and resources in the development of the software. It includes scheduling, milestones, reviews, staffing etc. Here the major goal is optimal usage of available resource in order to reduce the overall cost.
   c) **Software Configuration Management process**: - It's the process of understanding and formulating the necessary system requirements and establishing the exact amount and type of resource needed to successfully complete the software product.

2) **Process Management Processes: -** This process is responsible for monitoring every process that are occurring during the software development procedures. It ensures that high standards are followed during every process. It includes understanding the current process, analyzing its properties, determining possible improvements and then implementing the improvements in the processes.

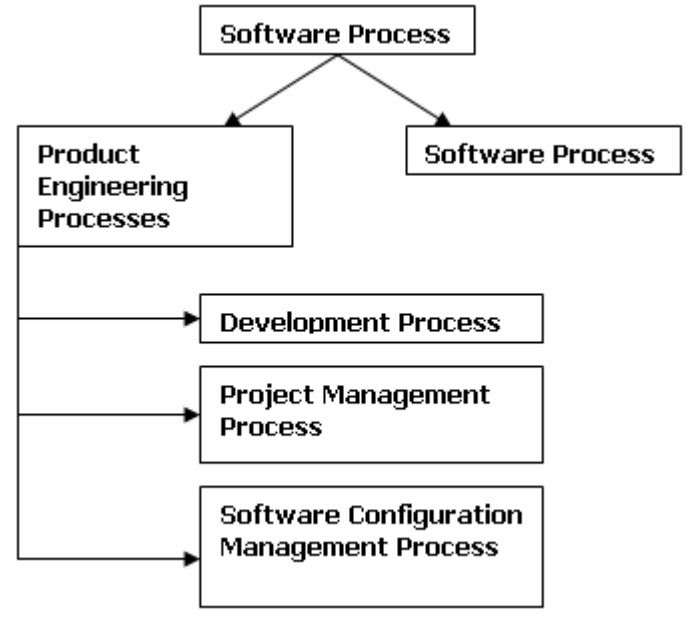


Fig Software Process

Characteristics of a Software Process

A software process should provide effective development, management and change management. Some of the desired characteristics are

1) **Predictability: -** It determines how accurately a process can give the desired outcomes if that particular process is followed and a prediction of the outcomes can be made before the product is actually completed. This is a fundamental property of any properly engineered process. The predictions could be in terms of size, cost, quality etc.
2) **Testability and Maintainability: -** A process is said to be proper if by following that process the final product be maintainable, also the cost on testing and maintenance should also reduce. Both testing and maintenance depend largely on the design and coding of software and these costs can be considerably reduced if the process that is being followed ensures that the coding is up to standards and flexible.

3) **Early defect removal and defect prevention: -** It's observed that errors occur throughout the development process. The cost of correcting errors of different phases is not the same and depends on when the error is detected and corrected. Detecting errors soon after they have been introduced is one of the major objectives of a software process.
4) **Process improvement: -** A good software process is such that it supports its own improvement. This is possible if the process itself provides some quantifiable data from time to time for its own evaluation and the data provided for evaluation should be able to show the strength and weakness in various stages of the process.

Software Metrics

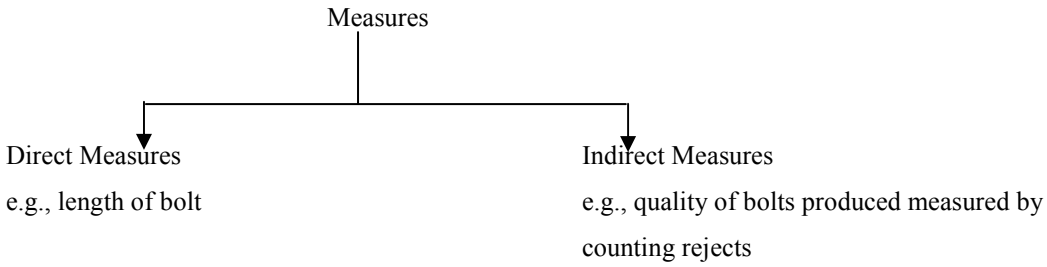
Software metrics refers to broad range of measures for computer software within the context of software project management. We are concerned with:

a) Productivity metrics output.
b) Quality metrics (customer requirements being satisfied or not)
c) Technical metrics (logically complexity of software, degree of modularity)

Why is software measured? Many reasons are there:

a) To indicate quality of product.
b) To access the productivity of people who produce the product.
c) To assess benefits (in terms of quality and productivity) derived from new software engineering methods and tools.
d) To form a baseline for estimation of cost, of resources and of schedules.
e) To help justify requests for new tools or additional training.

Types of Measures:



There are two types of measures:

1. **Direct Measures** – In software engineering process, following are the direct measures:
    a) Lines of Code (LOC)
    b) Execution Speed
    c) Memory Size
    d) Defects reported over some set period of time.
2. **Indirect Measures** – In software engineering process, following are the indirect measures:
    a) Functionality
    b) Quality
    c) Complexity
    d) Efficiency
    e) Maintainability

Note that one category of metrics is:

a) **Productivity Metrics** – That focuses on output of software engineering process.

b) **Quality Metrics** – That provides an indication of how closely software conforms to implicit and explicit customer requirements.

c) **Technical Metrics** – That focuses on character of software. E.g., its logical complexity, degree of modularity.

But there is another category of software metrics:

a) **Size oriented metrics** – Size oriented metrics are used to collect direct measures of software engineering output and quality.

b) **Function oriented metrics** – Function oriented metrics provide indirect measures and human oriented metrics collect information about the manner in which people develop computer software.

a) **Size oriented metrics** – It focuses on the size of the software that has been produced. If a software organization maintains simple records, a table of size oriented measures, such as shown below can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

Fig Size oriented metrics

It shows that for project: alpha, 12,100 LOC were developed with 24 person months of effort, at a cost of $168,000. Note that the effort and cost recorded in the table represent all software engineering activities (i.e. analysis, design, code and test) and not just coding. Furthermore, it shows that 365 pages of documentation were developed, 134 errors were recorded before software was released and 29 defects were encountered after release to the customer within first year of operation and three people worked on this project alpha.

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects4 per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development . Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

b) **Function-Oriented Metrics** – Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the ***function point***. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

			Weighting factor			
Measurement parameter	**Count**		**Simple**	**Average**	**Complex**	
Number of user inputs		×	3	4	6	=
Number of user outputs		×	4	5	7	=
Number of user inquiries		×	3	4	6	=
Number of files		×	7	10	15	=
Number of external interfaces		×	5	7	10	=
Count total	→					

Fig computing Function Point

Function points are computed by completing the table shown in Figure above. Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:

Number of user inputs – Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs – Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries – An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files – Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces – All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective. To compute function points (FP), the following relationship is used:

FP = count total * [0.65 + 0.01 * Σ(*Fi*)]

Where count total is the sum of all FP entries obtained from Figure above.

The *Fi* (*i* = 1 to 14) are "complexity adjustment values" based on responses to the following questions:

1. Does the system require reliable backup and recovery?

2. Are data communications required?

3. Are there distributed processing functions?

4. Is performance critical?

5. Will the system run in an existing, heavily utilized operational environment?

6. Does the system require on-line data entry?

7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?

8. Are the master files updated on-line?

9. Are the inputs, outputs, files, or inquiries complex?

10. Is the internal processing complex?

11. Is the code designed to be reusable?

12. Are conversion and installation included in the design?

13. Is the system designed for multiple installations in different organizations?

14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation and the weighting factors that are applied to information domain counts are determined empirically.

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

Advantages of FP

1. FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages.
2. It is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.

Disadvantages of FP

1. The method requires some "sleight of hand" in that computation is based on subjective rather than objective data.
2. The counts of the information domain (and other dimensions) can be difficult to collect after the fact.
3. FP has no direct physical meaning—it's just a number.

Measures for software Quality

There are many measures of software quality, correctness; maintainability, integrity, and usability provide useful indicators for the project team.

1. Correctness.

1. Correctness is the degree to which the software performs its required function.
2. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
3. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use.
4. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

2. Maintainability.

1. Maintainability is the ease with which a program can be
    - Corrected if an error is encountered,
    - Adapted if its environment changes, or
    - Enhanced if the customer desires a change in requirements
2. There is no way to measure maintainability directly; therefore, we must use indirect measures.
3. A simple time-oriented metric is ***mean-time-to change*** (MTTC), the time it takes to
    - Analyze the change request,
    - Design an appropriate modification,
    - Implement the change, test it, and
    - Distribute the change to all users.
    - On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.

3. Integrity.

1. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security.
2. Attacks can be made on all three components of software: programs, data, and documents.
3. To measure integrity, two additional attributes must be defined:

    a) Threat **b) Security.**

4. ***Threat*** is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time.
5. ***Security*** is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled.
6. The integrity of a system can then be defined as

    Integrity = summation [(1 – threat) * (1 – security)]

    Where threat and security are summed over each type of attack

4. Usability.

If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable.

Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

(1) The physical and or intellectual skill required to learn the system,

(2) The time required to become moderately efficient in the use of the system,

(3) The net increase in productivity measured when the system is used by someone who is moderately efficient, and

(4) A subjective assessment of user's attitudes toward the system.

Metrics for Source Code

Halstead's Theory: Halstead software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow:

Say, n_1 = The number of distinct operators that appear in a program

n_2 = The number of distinct operands that appear in a program

N_1 = The total number of operator occurances.

N_2 = The total number of operand occurances.

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language) and other features such as development effort, development time and the projected number of faults in the software.

Halstead shows that length N can be estimated using the formula given below:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

And program volume may be defined as follows:

$$V = N \log_2 (n_1 + n_2)$$

Where $N = N_1 + N_2$.

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume of the most compact form of a program to the volume of the actual program. In addition, L must always be less 1. In terms of premature measures, the volume ratio may be expressed as

$$L = 2/n_1 * n_1/n_2$$

Information Flow Metrics (IF – Metrics)

The IF – metrics is based on a premise that all systems (software) consist of components only that work together to influence the complexity of a system. System theory tells us that components that are highly coupled and that lack cohesion tend to be less reliable and less maintainable than those that are loosely coupled and are cohesive.

IF – metrics model the degree of cohesion and coupling for a particular system component. IF – metrics was applied to software systems by Henry and Kafura. They looked at UNIX OS and found a strong association between the IF – metrics and the level of maintainability applied to the components by developers.

IF – metrics are applied to the components of a system design. For e.g., for a component/ module – A, we can define 3 measures –

1. FAN – IN: A count of the number of other components that can call or pass control, to component – A.
2. FAN – OUT: It is the number of components that are called by component – A.
3. IF – metrics of component – A is

   IF (A) = [FAN – IN (A) * FAN – OUT (A)]

Q. 1) Computer the FP value for a project with the following information domain characteristics:s

No. of user inputs = 32

No. of user outputs = 60

No. of user inquiries = 24

No. of user files = 08

No. of External interfaces = 2

Assume that all complexity adjustment values are average. Assume 14 algorithms have been counted.

Sol: Given

No. of user inputs = 32

No. of user outputs = 60

No. of user inquiries = 24

No. of user files = 08

No. of External interfaces = 2

And complexity adjustment values are average, so,

Measurement Parameter	**Count**	**Weighting Factor**
No. of user inputs	32 * 4 =	128
No. of user outputs	60 * 5 =	300
No. of user inquiries	24 * 4 =	96
No. of user files	08 * 10 =	80
No. of External interfaces	2 * 7 =	14
Count total		618

So, FP = count total * [0.65 + 0.01 * Σ(*Fi*)]

Σ(*Fi*) = 14 * 3 = 42 (because in the question it is given that complexity adjustment values are average and all 14 algorithms are counted)

FP = 618 * [0.65 + 0.01 * 42]

FP = 661

Q. 2) Computer the FP value for a project with the following information domain characteristics:

Measurement Factor		Weighting Factor
No. of user inputs	40	4
No. of user outputs	50	5
No. of user inquiries	30	5
No. of user files	10	6
No. of External interfaces	5	10

Assume that all complexity adjustment values are average i.e. have a value of 3.

Q. 3) Compute FP for the following data set:

Inputs = 8

Outputs = 12

Inquiries = 4

Logical Files = 41

Interfaces = 1

Σ(*Fi*) = 41 (influence factor sum).

Software Cost Estimation

The major factors that influence software cost are

1. Programmer ability
2. Product complexity
3. Product size
4. Available time
5. Required reliability
6. Level of technology

Programmer Ability

It had been observed that on very large projects the differences in individual programmer ability will tend to average out and would not affect the project adversely. But on projects utilizing 5 or lesser programmers the factor of individual ability is significant and can affect the project positively or negatively. Hence individual programmer ability governs programmer productivity that affects the cost of the project.

Product Complexity

A software product can be classified on the basis of their complexity in to basic three types –

a) Application programs,

b) Utility programs,

c) Systems program.

Through extensive studies it had observed that Utility programs are 3 times as difficult to write as Application programs and that System programs are three times as difficult to write as Utility programs. So the levels of complexity can be stated as 1 - 3 - 9 for Application – Utility – Systems programs. The following equations were derived from the extensive research –

For calculating Programmer Months (PM): -

Application programs: - $PMap = 2.4 * (KDSI)^{1.05}$

Utility programs: - $PMup = 3.0 * (KDSI)^{1.12}$

Systems programs: - $PMsp = 3.6 * (KDSI)^{1.2}$

For calculating Development time for a program (TDEV): -

Application programs: - $TDEVap = 2.4 * (PM)^{1.05}$

Utility programs: - $TDEVup = 3.0 * (PM)^{1.12}$

Systems programs: - $TDEVsp = 3.6 * (PM)^{1.2}$

For calculating Average Staffing Level (SL): -

Application programs: - SLap= PMap / TDEVap

Utility programs: - SLup= PMup / TDEVup

Systems programs: - SLsp= PMsp / TDEVsp

Where, PM → Programmer Months and KDSI → Thousands of delivered source instructions.

A major consideration that needs to be taken in to account is extra coding required for housekeeping purpose. Housekeeping codes include that portion of coding that handles

a) Input/output,

b) Interactive user communication,

c) Human interface engineering and

d) Error checking & error handling.

So the estimation of cost on lines of code should include housekeeping codes.

Product Size

A large software product is obviously more expensive to develop than a small one. The equations derived for Programmer months (PM) and Development time (TDEV) show that the rate of increase in effort required grows with the number of source instructions at an exponential rate slightly greater than 1.

Available Time

Total project effort is sensitive to the calendar time available for the project completion. Software projects require more effort if the development time is compressed or expanded from the optimal time. After extensive research it had been observed that there is limit beyond which a software project cannot reduce its schedule even by buying more personnel and equipment. The limit occurs roughly at 75% of the nominal schedule.

Level of Technology

The level of technology provided by latest software products helps reducing overall cost. The type-checking and self-documentation aspects of high-level languages improve the reliability and modifiability of the programs created using these high-level languages. The familiarity of the programmer with the latest technology is also a contributing factor in effort reduction and cost reduction.

Software Cost Estimation Techniques

Cost estimates can be made either top-down or bottom-up.

Top-down estimation first focuses on system level costs, such as the computing resources and personnel required to develop the system, as well as the costs of

a) Configuration management,
b) Quality assurance,
c) System integration,
d) Training, and
e) Publications.

Personnel costs are estimated by examining the cost of similar past projects.

Bottom-up cost estimation first estimated the costs to develop each module or sub-system; those costs are combined to arrive at an overall estimate. Bottom-up estimation emphasizes the costs associated with developing individual system components, but may fail to account for system level costs, such as configuration management and quality control.

1) Expert Judgment Technique

The most widely used cost estimation technique is Expert judgment. Inherently it's a top-down estimation technique. Expert judgment relies on the experience background and business sense of one or more key people in the organization.

Groups of experts sometimes prepare a consensus estimate (collective estimate); this tends to minimize individual oversights and lack of familiarity with particular projects and neutralizes personal biases and the desire to win the contract through an overly optimistic estimate as seen in individual judgment style.

The major disadvantage of group estimation is the effect that interpersonal group dynamics may have on individuals in the group. Group members may not be candid enough due to political considerations. The presence of authority figures in the group or the dominance of an overly assertive group member.

2) Delphi cost estimation

This technique was developed to gain expert consensus without introducing the adverse side effects of group meetings. The Dephi can be adapted to software cost estimation in the following manners:-

A coordinator provides each estimator with the system definition document and a form for recording the cost estimation.

Estimators study the definition and complete their estimates anonymously. They may ask questions to the coordinator, but they do not discuss their estimates with one another.

The coordinator prepares and distributes a summary of the estimator's responses any unusual rationales noted by the estimators.

Estimators complete another estimate again anonymously, using the result from the previous estimate. Estimators whose estimates differ sharply from the group may be asked, anonymously, to provide justification for their estimates.

The process id iterated for as many rounds as required. No group discussion is allowed during the entire process.

3) The COCOMO Model

The Constructive Cost Model (COCOMO) is an empirical estimation model i.e., the model uses a theoretically derived formula to predict cost related factors. This model was created by "Barry Boehm". The COCOMO model consists of three models:-

1. **The Basic COCOMO model**: - It computes the effort & related cost applied on the software development process as a function of program size expressed in terms of estimated lines of code (LOC or KLOC).
2. **The Intermediate COCMO model**: - It computes the software development effort as a function of – a) Program size, and b) A set of cost drivers that includes subjective assessments of product, hardware, personnel, and project attributes.
3. **The Advanced COCOMO model**: - It incorporates all the characteristics of the intermediate version along with an assessment of cost driver's impact on each step of software engineering process.

The COCOMO models are defined for three classes of software projects, stated as follows:-

1. **Organic projects**: - These are relatively small and simple software projects which require small team structure having good application experience. The project requirements are not rigid.
2. **Semi-detached projects**: - These are medium size projects with a mix of rigid and less than rigid requirements level.
3. **Embedded projects**: - These are large projects that have a very rigid requirements level. Here the hardware, software and operational constraints are of prime importance.

In the following section we will discuss only the Basic and Intermediate COCMO models.

Basic COCOMO Model

The effort equation is as follows: -

$E = a * (KLOC)^{b}$

$D = c * (E)^{d}$

Where E → effort applied by per person per month, D → development time in consecutive months, KLOC → estimated thousands of lines of code delivered for the project. The coefficients a, b, and the coefficients c, d are given in the Table below

Software Project	A	b	c	D
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Intermediate COCOMO Model

The effort equation is as follows: -

$$E = a * (KLOC)^{b} * EAF$$

Where E → efforts applied by per person per month, KLOC → estimated thousands of lines of code delivered for the project, and the coefficients a and exponent b are given in the Table below. EAF is Effort Adjustment Factor whose typical ranges from 0.9 to 1.4 and the value of EAF can be determined by the tables published by "Barry Boehm" for four major categories – product attributes, hardware attributes, personal attributes and project attributes.

Software Project	a	b
Organic	3.2	1.05
Semidetached	3.0	1.12
Embedded	2.8	1.20

Work Breakdown Structures (WBS)

The WBS method is a bottom-up estimation tool.

It's a hierarchical chart that accounts for the individual parts of system.

There are two main classifications - a) Product hierarchy, b) Process hierarchy.

Product hierarchy: - It identifies the product components and indicates the manner in which the components are interconnected.

Process hierarchy: - It identifies the work activities and the relationship between them.

The primary advantage of the WBS technique are it helps in identifying and accounting the various process & product factors and in identifying the exact costs included in the estimates

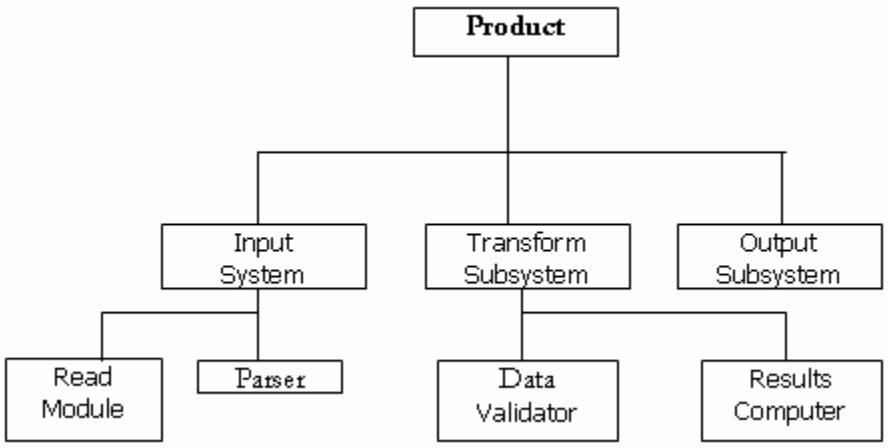


Fig Example of a Product Hierarchical Structure

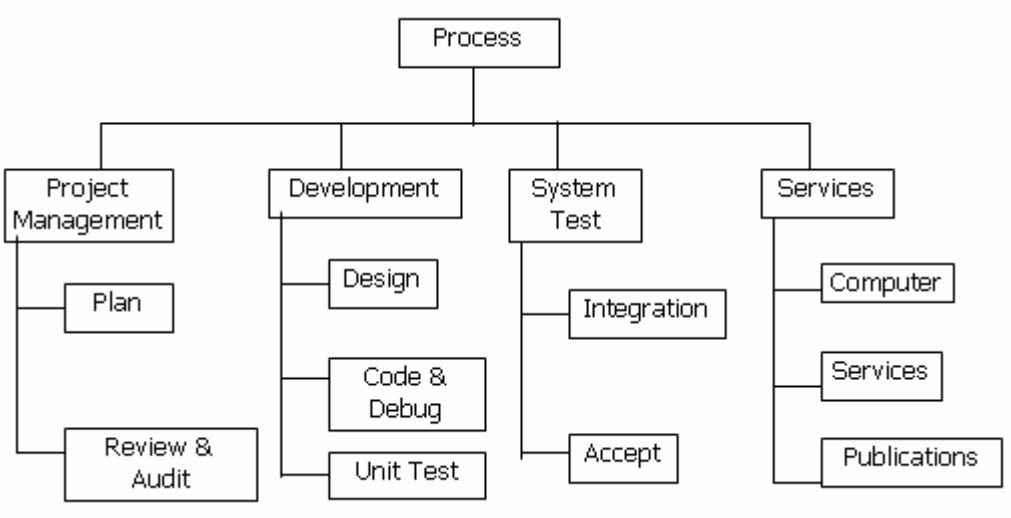


Fig Example of a Process Hierarchical Structure

Q.1) Consider a project to develop a full screen editor. The major components identified are screen edit, command language interpreter, file input and output, cursor movement, screen movement. The sizes for these are estimated to be 4K, 2K, 1K, 2K and 3K delivered source lines. Use COCOMO Model to determine:

a) Overall cost and schedule estimates.
b) Cost and schedules for different phases.

Sol:

a) Overall cost and schedule estimates using COCOMO Model:
For estimation we will go for Basic COCOMO model. The table for constants for Basic COCOMO Model is as follows:

Software Project	A	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

For our given project phases/ modules are there:

1. Screen Edit Size → 4K
2. Command Language Interpreter → 2K
3. File input and output → 1K
4. Cursor movement → 2K
5. Screen movement → 3K

Total size → 12K

Now, we will go for overall estimation of project:

If we are analyzing out project, then we find that full screen editor is a semidetached project. So, for estimation purpose, we will make, use of those constants.

$E = a * (KLOC)^b$

$D = c * (E)^d$

Where, E → Effort applied in person months.

D → Development time in chronological months.

$E = 3.0 * (12)^{1.12} = 48.50$ person months

$D = 2.5 * (48.50)^{0.35} = 9.72$ months.

Number of people N = E/D = 48.50 / 9.72 = 4.98 = 5 persons (approximately)

For completion of this project we will require 5 people.

(b) Now, we will go for estimation of each module:

All modules will be organic software project because they are simple and we will make use of concerned constants.

i. **Screen Edit** → 4K

$E = 2.4 * (4)^{1.05} = 10.28$ person months

$D = 2.5 * (10.20)^{0.38} = 6.06$ months

N = E/D = 1.6 = 2 persons (approximately)

ii. **Command Language Interpreter** → 2K

$E = 2.4 * (2)^{1.05} = 4.96$ person months

$D = 2.5 * (4.96)^{0.38} = 4.59$ months

N = E/D = 1.08 = 1 person (approximately)

iii. **File input and output** → 1K

$E = 2.4 * (1)^{1.05} = 2.4$ person months

$D = 2.5 * (2.4)^{0.38} = 3.48$ months

N = E/D = 0.68 persons

Not required dedicated individual, any person engaged with other module can do it simultaneously.

iv. **Cursor Movement** → 2K

$E = 2.4 * (2)^{1.05} = 4.96$ person months

$D = 2.5 * (4.96)^{0.38} = 4.59$ months

N = E/D = 1.08 = 1 person (approximately)

v. **Screen Movement** → 3K

$E = 2.4 * (3)^{1.05} = 7.6$ person months

$D = 2.5 * (7.6)^{0.38} = 5.4$ months

N = E/D = 1.4 = 1 person (approximately)

Q.2) Compute the estimation and duration of a project which is of type organic mode and estimated LOC is 10KLOC using COCOMO.

Sol: For estimation we will go for Basic COCOMO model. The table for constants for Basic COCOMO Model is as follows:

Software Project	A	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

For our given project estimated LOC = 10KLOC

Given mode is organic, so a = 2.4, b = 1.05, c = 2.5 and d = 0.38

Effort $E = a * (KLOC)^b$

$E = 2.4 * (10)^{1.05} = 26.93$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (26.93)^{0.38} = 8.74$ months

No of persons required N = E/D = 26.93/8.74 = 3.08 = 3 persons (approximately)

Q.3) Assume that the size of an organic type software product has been estimated to be 32,000 LOC and assume that the average salary of software engineer is Rs. 15,000 per month. Determine the effort required to develop the software product and the nominal development time.

Sol: For estimation we will go for Basic COCOMO model. The table for constants for Basic COCOMO Model is as follows:

Software Project	A	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

For our given project estimated LOC = 32KLOC

Given mode is organic, so a = 2.4, b = 1.05, c = 2.5 and d = 0.38

Effort $E = a * (KLOC)^b$

$E = 2.4 * (32)^{1.05} = 91$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (91)^{0.38} = 14$ months

Cost required to develop the product = 14 * 15,000 = Rs. 210,000

Q.4) Suppose that we are faced with developing a system such that we expect to have about 1,00,000 LOC. Compute the effort and development time for the organic and semidetached development mode.

Sol: For estimation we will go for Basic COCOMO model. The table for constants for Basic COCOMO Model is as follows:

Software Project	A	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

For our given project estimated LOC = 100 KLOC

a) **Organic Mode**: For organic mode, a = 2.4, b = 1.05, c = 2.5 and d = 0.38

Effort $E = a * (KLOC)^b$

$E = 2.4 * (100)^{1.05} = 302.14$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (302.14)^{0.38} = 21.89$ months

b) **Semidetached Mode**: For Semidetached mode, a = 3.0, b = 1.12, c = 2.5 and d = 0.35

Effort $E = a * (KLOC)^b$

$E = 3.0 * (100)^{1.12} = 521.34$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (521.34)^{0.35} = 22.33$ months

Q.5) The value of size of a program in KLOC and different cost drivers are given as size 300 KLOC, complexity 0.95, analyst capability 1.85, application of software engineering method 0.8, performance requirement 0.75. Calculate the effort for three types of projects i.e., organic, semidetached and embedded using COCOMO model.

Sol: For estimation we will go for intermediate COCOMO model. The table for constants for intermediate COCOMO Model is as follows:

Software Project	A	b	c	d
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

For our given project estimated LOC = 300 KLOC

Complexity → 0.95

Analyst Capability → 1.85

Application of software engineering method → 0.8

Performance Requirement → 0.75

So, EAF = 0.95 * 1.85 * 0.8 * 0.75 = 1.05

a) **Organic Mode** : For organic mode, a = 3.2, b = 1.05, c = 2.5 and d = 0.38

Effort $E = a * (KLOC)^b * EAF$

$E = 2.4 * (300)^{1.05} * 1.05 = 1340.7$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (1340.7)^{0.38} = 38.58$ months

No. of persons required N = E/D = 1340.7/38.58 = 34.75 = 35 persons (approximately)

b) **Semidetached Mode** : For Semidetached mode, a = 3.0, b = 1.12, c = 2.5 and d = 0.35

Effort $E = a * (KLOC)^b * EAF$

$E = 3.0 * (300)^{1.12} * 1.05 = 1873.6$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (1873.6)^{0.35} = 34.94$ months

No. of persons required N = E/D = 1873.6/34.94 = 53.62 = 54 persons (approximately)

c) **Embedded Mode** : For Embedded mode, a = 2.8, b = 1.20, c = 2.5 and d = 0.32

Effort $E = a * (KLOC)^b * EAF$

$E = 2.8 * (300)^{1.20} * 1.05 = 2759.9$ person months

Duration $D = c * (E)^d$

$D = 2.5 * (2759.9)^{0.32} = 31.55$ months

No. of persons required N = E/D = 2759.9/31.55 = 87.48 = 87 persons (approximately)

Scheduling

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a ***macroscopic schedule*** is developed. This type of schedule identifies all major software engineering activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a ***detailed schedule***. Here, specific software tasks (required to accomplish an activity) are identified and scheduled.

Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end-date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that

the end-date is set by the software engineering organization. Effort is distributed to make best use of resources and an end-date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second.

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification to software projects.

Program evaluation and review technique (PERT) and ***critical path method*** (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities:

- Estimates of effort
- A decomposition of the product function
- The selection of the appropriate process model and task set
- Decomposition of tasks

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project ***work breakdown structure*** (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow the software planner to

(1) Determine the *critical path*—the chain of tasks that determines the duration of the project;
(2) Establish "most likely" time estimates for individual tasks by applying statistical models; and
(3) Calculate "boundary times" that define a time "window" for a particular task.

Boundary time calculations can be very useful in software project scheduling. Slippage in the design of one function, for example, can retard further development of other functions. Riggs describes important boundary times that may be discerned from a PERT or CPM network:

(1) The earliest time that a task can begin when all preceding tasks are completed in the shortest possible time,
(2) The latest time for task initiation before the minimum project completion time is delayed,
(3) The earliest finish—the sum of the earliest start and the task duration,
(4) The latest finish— the latest start time added to task duration, and
(5) The total float—the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.

Both PERT and CPM have been implemented in a wide variety of automated tools that are available for the personal computer. Such tools are easy to use and make the scheduling methods described previously available to every software project manager.

Defining a Task Network

Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product(s).

A ***task network,*** also called an ***activity network,*** is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering tasks. Figure below shows a schematic task network for a concept development project.

The concurrent nature of software engineering activities leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, the planner must determine intertask dependencies to ensure continuous progress toward completion. In addition, the project manager should be aware of those tasks that lie on the critical path. That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule.

It is important to note that the task network shown in Figure below is macroscopic. In a detailed task network (a precursor to a detailed schedule), each activity shown in Figure below would be expanded.

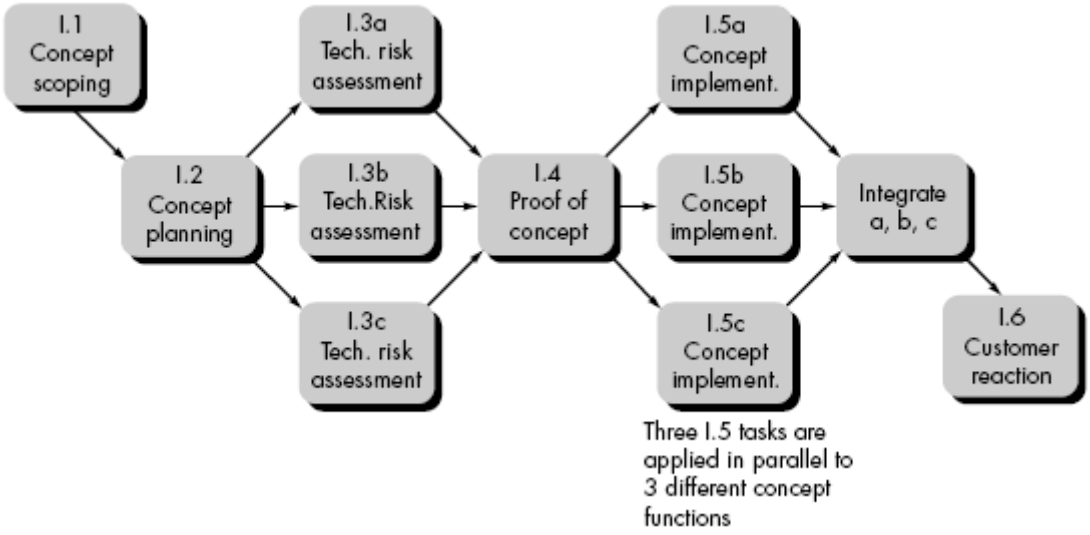


Fig A task set for concept development

Tracking the Schedule

Tracking can be accomplished in a number of different ways:

(1) Conducting periodic project status meetings in which each team member reports progress and problems.
(2) Evaluating the results of all reviews conducted throughout the software engineering process.
(3) Determining whether formal project milestones have been accomplished by the scheduled date.
(4) Comparing actual start-date to planned start-date for each project task listed in the resource table.
(5) Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
(6) Using earned value analysis to assess progress quantitatively.

Earned Value Analysis

To determine the earned value, the following steps are performed:

1. The ***budgeted cost of work scheduled*** **(BCWS)** is determined for each work task represented in the schedule. During the estimation activity, the work (in person-hours or person-days) of each software engineering task is planned.

2. The BCWS values for all work tasks are summed to derive the budget at completion, BAC. Hence,

$$BAC = \Sigma (BCWS_k) \text{ for all tasks } k$$

3. Next, the value for ***budgeted cost of work performed*** (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Defect Removal Efficiency (DRE)

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$DRE = E/(E + D)$$

Where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task. For example, the requirements analysis task produces an analysis model that can be reviewed to find and correct errors. Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

$$DRE_i = E_i/(E_i + E_{i+1})$$

Where E_i is the number of errors found during software engineering activity i and E_{i+1} is the number of errors found during software engineering activity $i+1$ that are traceable to errors that were not discovered in software engineering activity i. A quality objective for a software team (or an individual software engineer) is to achieve DRE_i that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

Error Tracking

Error tracking allows you to compare current work with past efforts and provides a quantitative indication of the quality of the work being conducted. Any errors that are not uncovered (but found in later tasks) are considered to be defects, D. Defect removal efficiency has been defined as

$$DRE = E/(E + D)$$

DRE is a process metric that provides a strong indication of the effectiveness of quality assurance activities.

Risk Management

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not.

Steps of Risk Analysis and Management:-

1. Recognizing what can go wrong is the first step, called "risk identification."
2. Each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur.
3. Risks are ranked, by probability and impact.
4. Finally, a plan is developed to manage those risks with high probability and high impact.

In short, the four steps are:

1. Risk identification
2. Risk Projection
3. Risk assessment
4. Risk management

Risk always involves two characteristics a set of risk information sheets is produced.

1. Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks.

2. Loss—if the risk becomes a reality, unwanted consequences or losses will occur.

Types of risks that are we likely to encounter as the software is built

1. Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.

2. Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible.

3. Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are

- Market risk,
- Strategic risk,
- Management risk, and
- Budget risks.

4. Known risks are those that can be uncovered after careful evaluation of the project plan.

5. Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

6. Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Reactive risk strategies:-

1. Reactive risk strategies follows that the risks have to be tackled at the time of their occurrence.
2. No precautions are to be taken as per this strategy.
3. They are meant for risks with relatively smaller impact.

Proactive risk strategies:-

1. Proactive risk strategies follows that the risks have to be identified before start of the project.
2. They have to be analyzed by assessing their probability of occurrence, their impact after occurrence, and steps to be followed for its precaution.
3. They are meant for risks with relatively higher impact.

Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

One method for identifying risks is to create a **risk item checklist.** The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic categories:

- ***Product size***—risks associated with the overall size of the software to be built or modified.
- ***Business impact***—risks associated with constraints imposed by management or the marketplace.
- ***Customer characteristics***—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- ***Process definition—***risks associated with the degree to which the software process has been defined and is followed by the development organization.
- ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.
- ***Technology to be built***—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- ***Staff size and experience***—risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk Projection

Risk projection also called ***risk estimation****,* attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities:

(1) Establish a scale that reflects the perceived likelihood of a risk,
(2) Delineate the consequences of the risk,
(3) Estimate the impact of the risk on the project and the product, and
(4) Note the overall accuracy of the risk projection so that there will be no misunderstandings.

Risk Assessment

At this point in the risk analysis process we have established a set of triplets of the form :

[ri,li,xi]

Where ri is risk, li is the likelihood (probability) of the risk, and xi is the impact of the risk.

During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to rank the risks that have been uncovered, and begin thinking about ways to control and/or avert risks that are likely to occur.

Therefore, during risk assessment, we perform the following steps:

1. Define the risk referent levels for the project.
2. Attempt to develop a relationship between each (ri, li, xi) and each of the referent levels.

3. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
4. Try to predict how compound combinations of risks will affect a referent level.

Risk Mitigation, Monitoring, and Management

1. An effective strategy must consider three issues:
    a) Risk avoidance
    b) Risk monitoring
    c) Risk management and contingency planning
2. High staff turnover in any organization will have a critical impact on project cost and schedule.
3. To mitigate the risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are
    - Meet with current staff to determine causes for turnover • Mitigate those causes that are under our control before the project starts.
    - Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
    - Organize project teams so that information about each development activity is widely dispersed.
    - Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
    - Conduct peer reviews of all work
    - Assign a backup staff member for every critical technologist.

Computer Aided Software Engineering (CASE)

Computer Aided Software Engineering (CASE) is a set of automated tools for implementing software engineering activities and standards during software development process. Following is a brief overview of CASE architecture.

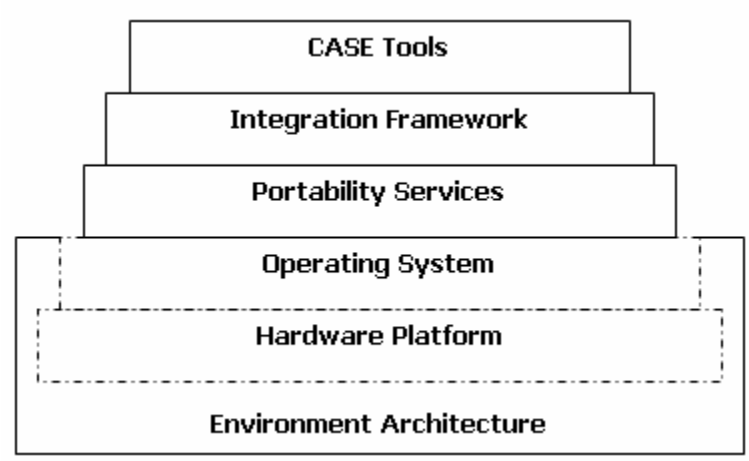


Fig CASE Building Blocks

1) The environment architecture is composed of the hardware platform and operating system support which includes networking and database management software, this forms the basic ground for CASE.
2) A set of portability services provides a bridge between CASE tools and their integration framework along with the environment architecture.
3) The integration framework is a collection of specialized programs that enable individual CASE tools to communicate with each other, to create a project database and to exhibit the same lookup for the software engineer as he would feel when doing the software engineering procedures manually.
4) Portability services provide CASE tools and their integration framework to become portable and migrate along different hardware platforms and operating systems without much change.

The CASE framework provides a very large range of tools for conducting all the possible software engineering activities. Here is a listing of some of the commonly used tools provided by a CASE framework :-

1) **Information Engineering Tools**:-These tools model business information as it moves between various organizations. They represent –
    a) Business data objects,
    b) Relationships between various business data objects and
    c) The flow of business data objects between organizations.
2) **Process Modeling and Management Tools**:- These tools provide –
    a) Key elements of a process for better understanding of the process,
    b) Links to the description of work tasks related to a process and
    c) Links to other related process management tools.
3) **Project Planning Tools**:- They provides tools related to –
    a) Software project effort and cost estimation,
    b) Project scheduling.
4) **Risk Analysis Tools**:- They provide information and methods to a project manager in the following areas –
    a) Building risk analysis tables, and
    b) Provide guidance for risk identification and analysis.
5) **Project Management Tools**:- These tools provides a project manager the following help –
    a) Creating, tracking and monitoring project plans and schedules, and
    b) Providing metrics to estimate & evaluate project and product quality factors.
6) **Documentation Tools**:- These tools help in the following manner –
    a) Document production, and
    b) Desk-top publishing of the documents.
7) **System software Tools**: -These tools include –
    a) E-mails,
    b) Electronic bulletin boards, and
    c) Network communication software.
8) **Software Configuration Management Tools** :- These tools help in the following areas of SCM –
    a) Identification,
    b) Version control,
    c) Change control,
    d) Auditing and
    e) Status accounting.
9) **PRO / SIM Tools**: - These are a set of Prototyping and Simulation tools for predicting the behavior of a real-time system before its production. They help the software engineers to build prototypes of the systems for analyzing the functionality, operation and response of the system.
10) **Test Management Tools**: - These tools help in the process of control and coordination for software testing procedures. It has a Test Driver that reads one or more test cases from a testing file, formats the test data to fit to the needs of the software under test, and then invokes the software to be tested.