Table of Contents

1. SOFTWARE ENGINEERING

Engineering: Engineering is the application of scientific, economic, social, and practical knowledge in order to invent, design, build, maintain, research, and improve structures, machines, devices, systems, materials and processes.

Software: Software is i) instructions(computer programs) that when executed provide desired function and performance. ii) data structures that enable the programs to adequately manipulate information and iii) documents that describe the operation and use of the programs.

Software is a general term for the various kinds of *program*s used to operate computers and related devices.

Software Engineering**:** *IEEE* defines software engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

SOFTWARE CHARACTERISTICS

- **Software is developed or engineered, it is not manufactured**
  Unlike hardware, software is logical rather than physical. It has to be designed well before producing it. In spite of availability of many automated software development tools, it is the skill of the individual, creativity of the developers and proper management by the project manager that counts for a good software product.
- **Software does not "wear out"**
  As time progresses, the hardware components start deteriorating-they are subjected to environmental maladies such as dust, vibration, temperature etc. and at some point of time they tend to breakdown. The defected components can then be traced and replaced. But, software is not susceptible to the environmental changes. So, it does not wear out. The software works exactly the same way even after years it was first developed unless any changes are introduced to it. The changes in the software may occur due to the changes in the requirements. And these changes may introduce some defects in it thus, deteriorating the quality of software, so software need to maintain properly.
- **Most software is custom-built, rather than being assembled from existing components**
  Most of the engineered products are first designed before they are manufactured, Designing includes identifying various components for the product before they are actually assembled. Here several people can work independently on these components, thus making the manufacturing system highly flexible. In software, breading a program into modules is difficult task, since each module is highly interlinked with other modules. Further, it requires lot of skill to integrate different modules into one. Now a days the term component is widely used in software industry where object oriented system is in use.

CATEGORIES OF SOFTWARE

Software are mainly classified into two. **System software and Application software**.

System Software: System software is a set of programs which manages and controls the internal operations of a computer system. It is a collection of programs which is responsible for using computer resources efficiently and effectively. That is a collection of programs written to service other programs. The system software is again categorized as **Operating systems and Language processors.**

***Operating System (OS*)**: OS is considered as the fundamental system software. OS is an interface between the user and hardware devices. OS provides services like hardware control, memory management, multitasking etc.

Example: Windows 8, Windows 7, Windows XP, Ubuntu, Mac OS, Android, DOS, UNIX, Google OS etc.

Language Processors (LP): Computer can recognize only the machine language/ Binary language (0s and 1s). But programmers used to write programs in High Level Language (HLL) which is easily understandable to the humans. So there is a need of a translator which converts the programs written in HLL into an equivalent machine language. The system programs which performs this translation is called Language Processors. The different language processors are

i) **Assembler:** Coverts the program written in assembly language into its equivalent machine language.

ii) **Interpreter:** Converts the program written in High Level Language (HLL) into its equivalent machine language by converting and executing each line by line. If an error is there in the HLL code, it reports the error and execution terminates. Execution is resumed only after correcting the error in that line.

iii) **Compiler:** Compiler is similar to Interpreter which translates a High Level Language into its equivalent machine language. But Compiler compiles a whole program and then it lists out the errors along with its line numbers. If there are no errors in the HLL code, computer generates the object files. The process of translating a HLL into its equivalent machine language is termed as **compilation.**

Application Software: Application software are programs which are developed in order to perform a particular task or functionality. That is they are stand - alone programs that solve a specific business need. Application software are programs which are designed to run under an operating system. They range from word processors and Internet browsers to video games and media players. Application software is further classified into three categories.

i) Packages ii) Utilities iii) Customized Software

Packages: Application package software, or simply an application package, is a collection of software programs that have been developed for the purpose of being licensed to third-party organizations. Application packages are generally designed to support commonly performed business functions and appeal to multiple types of user organizations. Various packages are listed below.

- Word Processing Software: Allows users to create, edit a document.
  Example: MS Word, Word Pad, Note pad etc.
- Spreadsheet Software: Allows users to create document and perform calculation. Example: MS Excel, Lotus1-2-3 etc.
- Database Software: Allows users to store and retrieve vast amount of data. Example: MS Access, MySQL, Oracle etc.
- Presentation Graphic Software: Allows users to create visual presentation. Example: MS Power Point
- Multimedia Software: Allows users to create image, audio, video etc.
  Example: Real Player, Media Player etc.

Utilities: Utility software is a collection of one or more programs that helps the user in system maintenance tasks and in performing tasks of routine nature. Utility programs help the users in disk formatting, data compression, data backup, scanning for viruses etc. It improves the efficiency and performance of the computer.

Examples of utility software are:

- Anti-virus
- Registry cleaners
- Disk defragmenters
- Data backup utility
- Disk cleaner

Customized software: Custom software (also known as **bespoke software** or **tailor-made software**) is software that is specially developed for some specific organization or other user. Large companies commonly use custom software for critical functions, including content management, inventory management, customer management, human resource management etc.

Hierarchy from tokens to software:

Tokens → Instructions / LOC → Modules / Functions → Programs → Software → Packages

A package is a collection of various software. A software consists of various programs. A program may contain various functions or modules. Each module or function may have multiple instructions or Line Of Code (LOC). An instruction consist of various tokens.

2. SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

Life cycle model: A life cycle model describes the different activities that need to be performed to develop a software. It also shows the sequencing of these activities.

SDLC: A SDLC is a descriptive and diagrammatic representation of the software life cycle.

Need of SDLC:

- Encourages development of software in a systematic and disciplined manner.
- Provides precise understanding among team members, thus helps to avoid chaos during development phases.
- Reduces project failure
- Makes easier to allocate resources and persons during various phases of software development.

WATERFALL MODEL / CLASSIC LIFE CYCLE MODEL / LINEAR SEQUENTIAL MODEL

The first published model of software development process was derived from other engineering processes. The classical waterfall model is the most obvious way to develop software. But it is not a practical model because it cannot be used in actual software development. So Waterfall model is a theoretical way of developing software. Then why it is relevant because the various other life cycle model is based on this Waterfall model. Waterfall model is also called as **Linear sequential model or Classic life cycle model.** The various phases of Waterfall model is illustrated below.

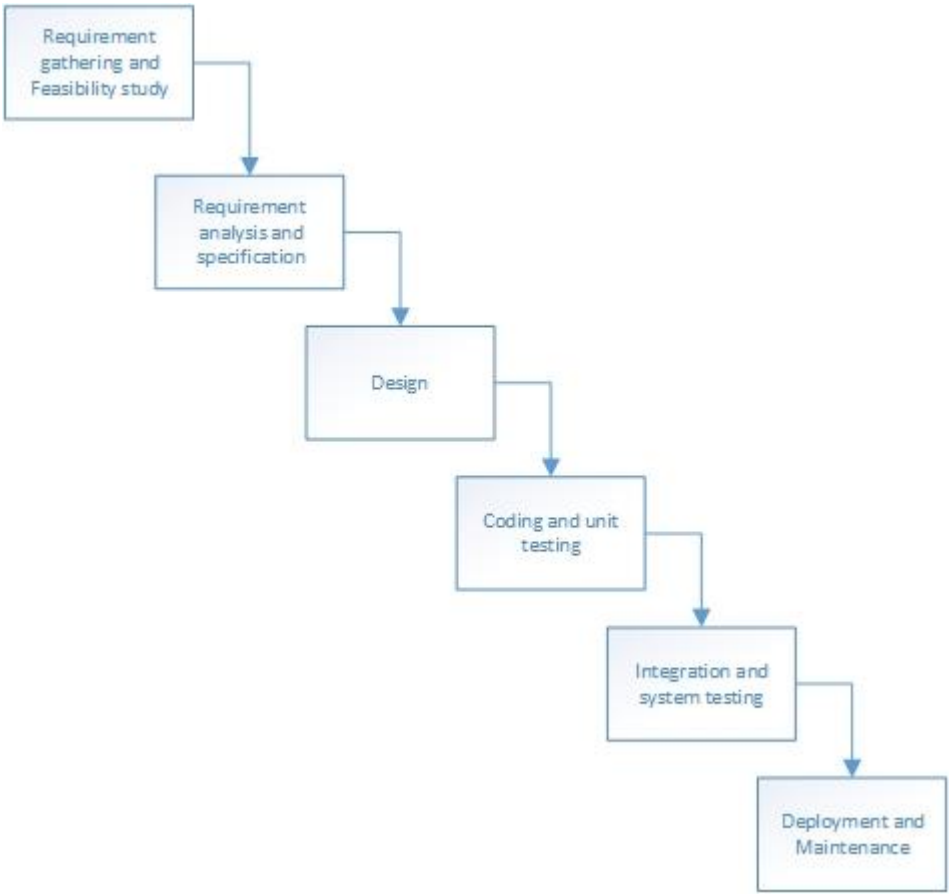


Figure 2.1: Waterfall model

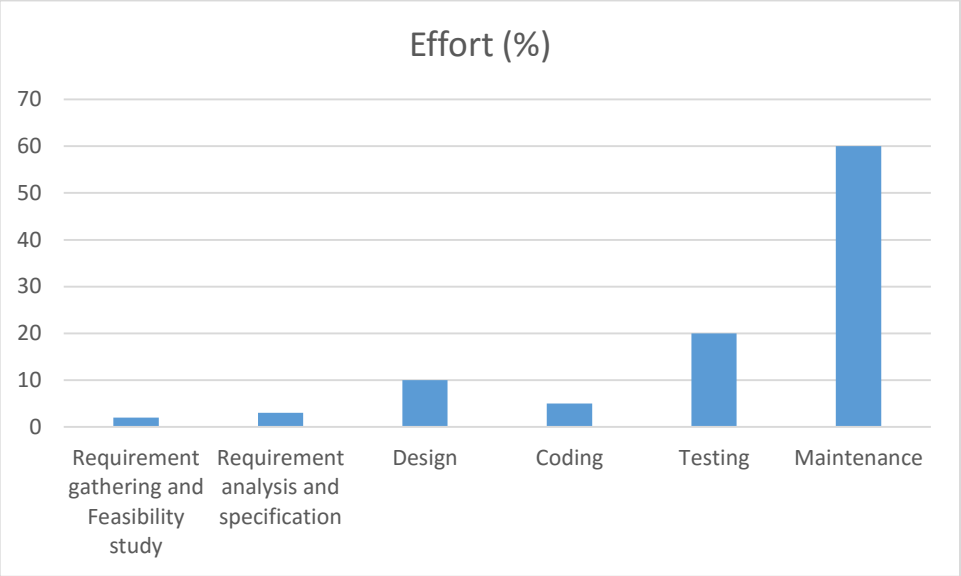
- The basic principle in this model is, dividing the entire life cycle into various phases. Each phase has a well -defined function or characteristics. In Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.
- No feed backing is provided between phases. That is, it is not possible to go back to a previous phase and make changes.
- **Requirement gathering:** This phase involves understanding what you need to design and what its intended purpose or function is. The requirements may be collected directly from the customer / client or by questionnaire survey or by visiting the target environment where the software is being deployed.
- **Feasibility study:** After identifying the various requirements feasibility study is conducted. The main aim of feasibility study is to determine whether it would be practically possible to develop the product. The identified requirements should be technically, economically and operationally feasible.
    1) *Technical feasibility*: Ensuring whether the requirements can be implemented using the existing technology or not.
    2) *Economic feasibility*: Ensuring whether the requirements can be implemented within the given budget or not.
    3) *Operational feasibility*: Focuses on the degree to which the proposed development projects fits in with the existing business environment and objectives with regard to development schedule, delivery date, corporate culture, and existing business processes.
- **Requirement analysis:** As per the requirements, the software and hardware needed for the proper completion of the project is analysed in this phase. Right from deciding which computer language should be used for designing the software, to the database system that can be used for the smooth

functioning of the software, such features are decided at this stage. The main activities in this phase are identifying inconsistent requirements and prioritizing the requirements.

- **Requirement specification:** During requirement specification, the requirements are documented in a suitable format. The document used for this purpose is called SRS (Software Requirement Specification). The components of SRS are
    1) *Functional requirements:* Specify the expected behaviour of the system - which outputs should be produced from the given inputs
    2) *Non – functional requirements:* Specifies the performance constraints on the software system.
    3) *Goal of implementation:* Specifies about the need of implementing the software and mentioning its application areas.
- **Design:** The goal of design phase is to transform the requirements mentioned in the SRS document into a structure that is suitable for implementation in some programming languages. The algorithm (pseudo-code) of the program or the software code to be written in the next stage, is created now. This algorithm forms the backbone for the actual coding process. Proper planning relating to the design of user interface, flowcharts is done here. Design process is also documented to form the *design document* of the software. The activities carried out during design phase are
    i) Developing algorithms to implement the various requirements / functionalities.
    ii) Developing flow charts, DFDs, Structure charts, UML diagrams etc.
    iii) Choosing the appropriate data structures.
    iv) Choosing the suitable design methodology with architectural design and detailed design.
- **Coding:** Based on the algorithm or flowchart designed, the actual coding of the software is carried out at this stage. The flowcharts / algorithms are converted into instructions written in a programming language.
- **Testing:** The software designed, needs to go through constant software testing and error correction processes to find out if there are any flaw or errors. Testing is done so that the client does not face any problem during the installation of the software. Testing is done with the intention to find errors. Testing is also documented to form *Test reports*.
- **Unit testing:** In modular programming methodology, the software is divided into various modules or units. A unit is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. Usually performed by developers itself.
- **Integration and System testing:** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. System testing usually consist of three different testing methodologies.
    i) Alpha testing: Testing performed by the developing team itself.
    ii) Beta testing: Testing performed by a friendly set of customers.
    iii) Acceptance testing: Testing performed by the customer after the product delivery to determine whether to accept or reject the delivered product.
- **Deployment:** It mainly refers the product release, installation of product in the target environment, activating the product, Updating etc.
- **Maintenance:** Modification of a software product after delivery to correct faults, to improve performance or other attributes. Maintenance activities are classified as follows:
    i) *Corrective maintenance*: Correcting the errors that are not identified during the product development phase.

ii) *Perfective maintenance*: Improving the implementation of the system and enhancing the system functionalities according to the client's requirements.
iii) *Adaptive maintenance*: Usually done for porting the software to work in a new environment (For example, new Operating system or new hardware platform etc.)
iv) *Preventive maintenance*: Implementing changes to prevent the occurrence of errors.

- **Advantages / Pros of Waterfall model**
  i) Relatively simple to understand
  ii) Easier to allocate resources and persons.
  iii) Each phase of development proceeds sequentially.
- **Disadvantages / Cons of Waterfall model**
  i) Requirements need to be specified before the development proceeds.
  ii) No feedback between phases. Hence not applicable in projects where requirements are changing rapidly.
  iii) Do not involve risk management.
  iv) No scope for error correction
  v) Highly impractical for most projects because only few projects can be divided into such water-tight phases.
- **Applications**
  It can be applied in long term projects or the projects where requirements are finite.
- **Effort Distribution among different phases**



ITERATIVE WATERFALL MODEL

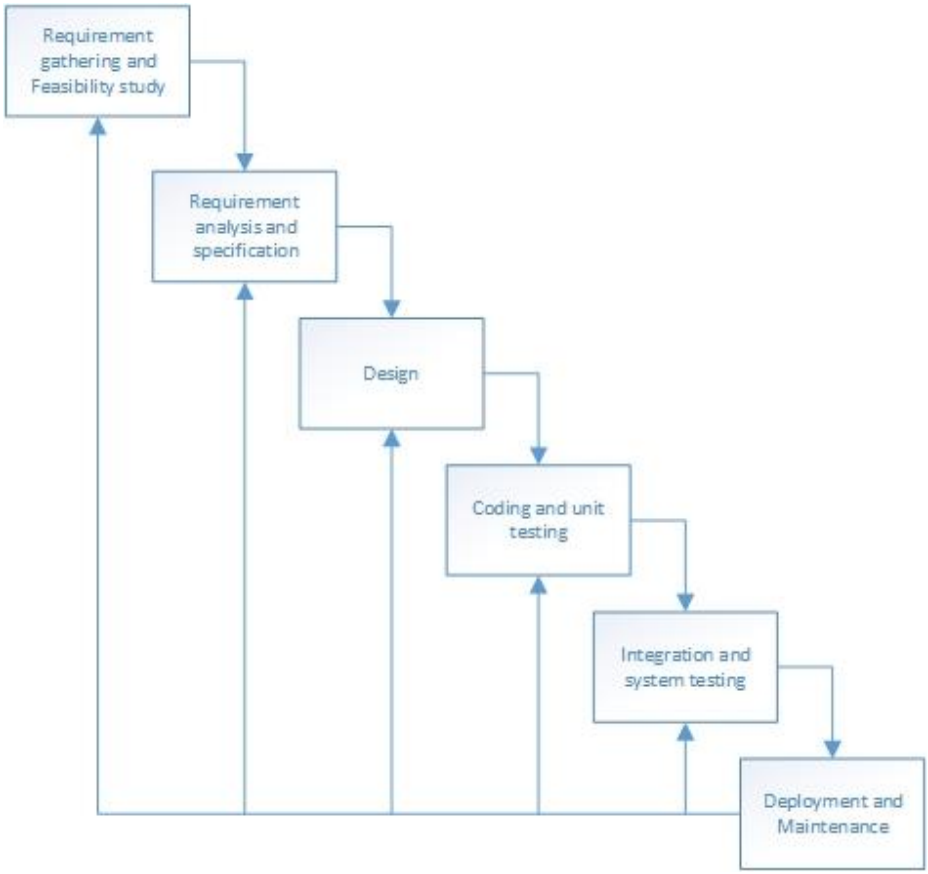


Figure: Iterative Waterfall model

Classical Waterfall model is an idealistic model. So we made necessary changes to the classical waterfall model, so that it becomes applicable to practical software development projects. Essentially the main change in Iterative waterfall model is in the form of providing feedback paths from every phase to its preceding phases as shown in the above figure. The feedback paths allow for correction of the errors committed during a phase, as and when these are detected in the later phase.

DISADVANTAGES /CONS:

- Cannot handle different types of risks evolved during development.
- Difficult to follow a rigid phase sequence. *That is once a team member complete his work earlier, he would have to be idle for other members to complete their work, before proceeding into his new activity.*

PROTOTYPING MODEL

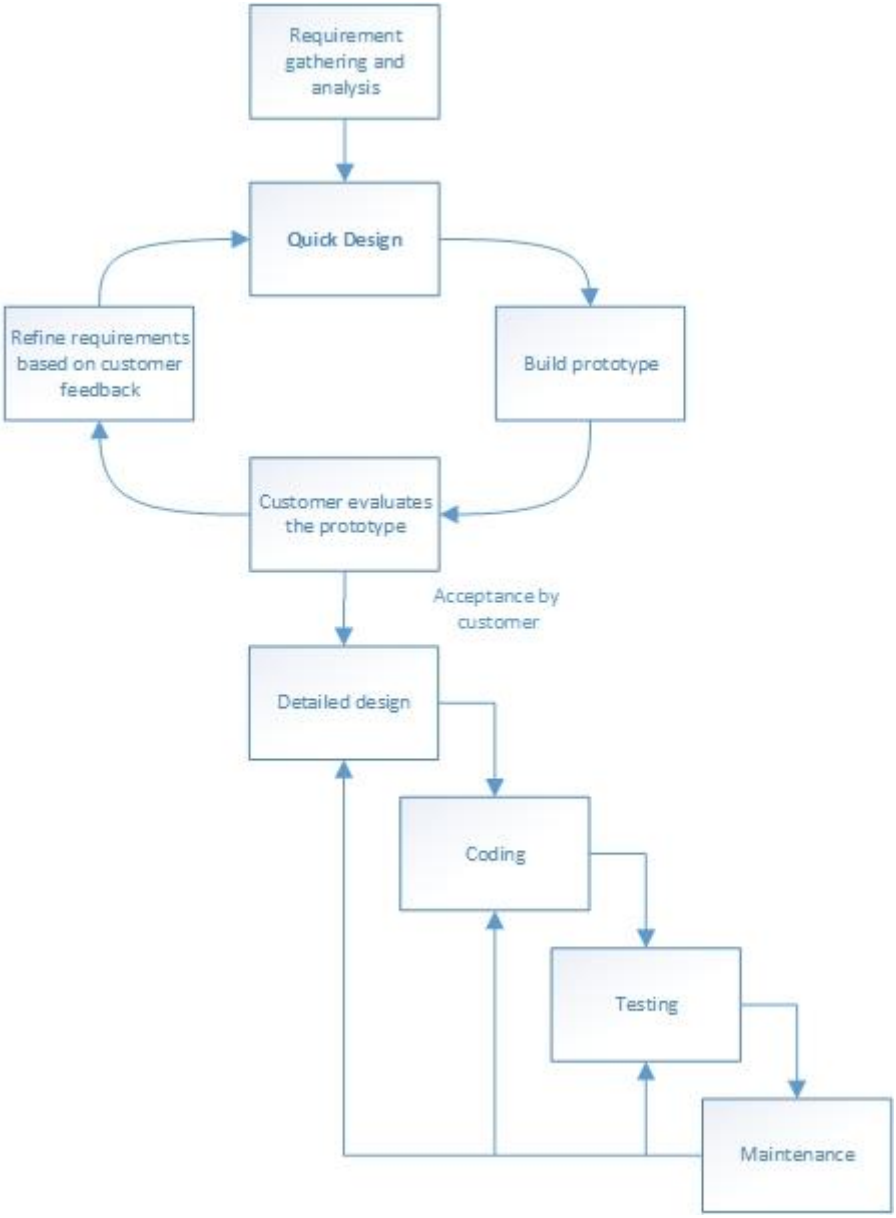


Figure: Prototyping model

- Prototyping model requires that before carrying out the development of the actual software, a working prototype of the system should be built. A prototype is a toy implementation or dummy model of the actual system. It is built using several shortcuts like dummy functions, GUI outlook using buttons etc.
- A prototype model has limited functional capabilities, low reliability and inefficient performance as compared to the actual software. The prototype gives the user an actual feel of the system.
- In this model it is assumed that all the requirements may not be known at the start of the development of the system.
- This model allows the users to interact and experiment with a working model of the system known as prototype.
- The code for the prototype is usually thrown away.
- Prototype can be created by the following approaches:
  i) By creating the main user interfaces without any substantial coding so that users can get a feel of how the actual system will appear.

ii) By abbreviating a version of the system that will perform limited subsets of functions.

iii) *By using system components to illustrate the functions that will be included in the system to be developed.*

- **Execution of prototyping model:** Prototyping begins with an initial requirement gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for his evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype. Once the customer approves the prototype, the actual system is developed using the iterative waterfall approach.

PHASES IN PROTYPING MODEL

- **Requirements gathering and analysis:** This phase involves understanding what you need to design and what its intended purpose or function is. The requirements may be collected directly from the customer / client or by questionnaire survey or by visiting the target environment where the software is being deployed. The main activities in requirement analysis phase are identifying inconsistent requirements and prioritizing those requirements. After that a suitable SRS document is prepared.
- **Quick design:** When requirements are known, a preliminary design or quick design for the system is created. It is not a detailed design and includes only the important aspects of the system, which gives an idea of the system to the user. A quick design helps in developing a prototype.
- **Build prototype:** Information gathered from quick design is modified to form the first prototype, which represents the working model of the actual system. It is a dummy model or a toy implementation of the actual software. It provides the look and feel of the actual system with limited functionalities.
- **Customer evaluation / User evaluation:** The prototype is presented before the user for proper evaluation in order to recognize the strengths and weaknesses such as what is to be added or removed. Comments and suggestions are collected from the users and are provided to the developer.
- **Refining prototype:** Based on the customer suggestions, the current prototype is refined. That is a new prototype is developed, incorporating the customer suggestions. The new prototype is presented to the user for evaluation. Customer evaluated it and provide modifications. Based on that, again the model is refined. This process continues until customer satisfied with the developed prototype. A final system is developed on the basis of the final prototype.
- **Detailed Design:** The goal of this phase is to convert the prototype model into target system with a detailed design. The activities carried out during this phase are
    i) Developing algorithms to implement the various requirements / functionalities.
    ii) Choosing the appropriate data structures.
- **Coding:** Based on the algorithm or flowchart designed, the actual coding of the software is carried out at this stage. The flowcharts / algorithms are converted into instructions written in a programming language.
- **Testing:** The software designed, needs to go through constant software testing and error correction processes to find out if there are any flaw or errors. Testing is done so that the client does not face any problem during the installation of the software. Testing is done with the intention to find errors. Testing is also documented to form *Test reports*.
- **Unit testing:** In modular programming methodology, the software is divided into various modules or units. A unit is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. Usually performed by developers itself.

- **Integration and System testing:** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. System testing usually consist of three different testing methodologies.
    i) Alpha testing: Testing performed by the developing team itself.
    ii) Beta testing: Testing performed by a friendly set of customers.
    iii) Acceptance testing: Testing performed by the customer after the product delivery to determine whether to accept or reject the delivered product.
- **Maintenance:** Modification of a software product after delivery to correct faults, to improve performance or other attributes. Maintenance activities are classified as follows:
    i) *Corrective maintenance*: Correcting the errors that are not identified during the product development phase.
    ii) *Perfective maintenance*: Improving the implementation of the system and enhancing the system functionalities according to the client's requirements.
    iii) *Adaptive maintenance*: Usually done for porting the software to work in a new environment (For example, new Operating system or new hardware platform etc.)
    iv) *Preventive maintenance*: Implementing changes to prevent the occurrence of errors.
- **ADVANTAGES / PROS:**
    - ✓ Improved customer satisfaction.
    - ✓ Developer gains experience from developing prototypes which results in the better implementation of requirements.
    - ✓ Chance of occurring errors is comparatively less.
- **DISADVANTAGES /CONS:**
    - ✓ Model is time consuming and expensive.
    - ✓ Developer may compromise with the quality of the software
    - ✓ Prototyping can lead to false expectations. For example, a situation may be created where the user believes that the development of the system is finished when it is not.
- **APPLICATIONS**
    - ✓ Used in the development of systems where GUI has much importance.
    - ✓ Prototype model should be used when the desired system needs to have a lot of interaction with the end users. Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model.
    - ✓ Excellent for designing good human computer interface systems.
    - ✓ Can be used when the technical solution is not clear for the developer.

SPIRAL MODEL

- Spiral model was proposed Barry Boehm in 1986. This model combines the best features of the prototyping model and waterfall model and is advantageous for large, complex and expensive projects.
- IEEE defines the Spiral model as '*a model of the software development process in which the constituent activities, typical requirements analysis, preliminary and detailed design, coding, integration and testing are performed iteratively until the software is complete*'.
- The objective of Spiral model is to emphasize management to evaluate and resolve risks in the software projects.
- Different risks in the software project are project overruns, changed requirements, loss of key project personnel, delay of necessary hardware, competition with other software developers and technological breakthroughs etc.
- Rather than represent the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral. Each loop in the spiral represents a phase

of the software process. Thus the inner most loop might be concerned with system feasibility, the next loop with system requirements definition, the next loop with system design and so on.

- Each loop in the Spiral model is divided into four quadrants. Each quadrant has a well-defined set of activities. The four quadrants are:
  i) Determine objectives, alternatives and constraints **(Objective setting)**
  ii) Evaluate alternatives, identify and resolve risks. **(Risk assessment and reduction)**
  iii) Develop and verify next – level product **(Development and validation)**
  iv) Plan the next phase. **(Planning)**

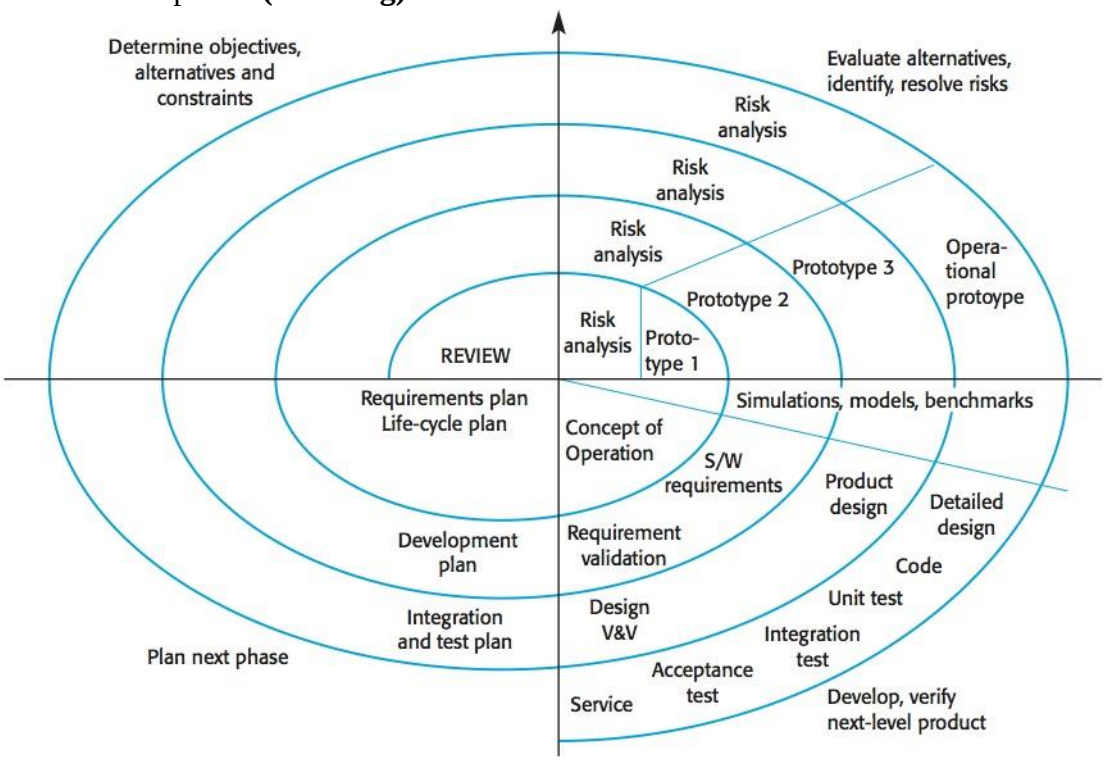


Figure: Spiral model

PHASES OR QUADRANTS IN SPIRAL MODEL

- **Determine objectives, alternatives and constraints:** Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
- **Evaluate alternatives, identify and resolve risks**: For each of the identified risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
- **Develop and verify next – level product:** After risk evaluation, a development model for the system is then chosen. Risk management considers the time and effort to be devoted to each project activity such as planning, configuration management, quality assurance, verification and testing.
- **Plan the next phase:** The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.
- One of the key features of Spiral model is that each cycle is completed by a review conducted by the individuals or users. This includes review of all the intermediate products, which are developed during the cycles.

- The radius of the spiral represents the cost of development, whereas the angular displacement represents the progress in development.
- The important distinction between Spiral model and other software process models is the explicit consideration of risk in the spiral model. Informally, Risk is something which can go wrong.
- There are no fixed phases such as specification or design in the spiral model. The spiral model encompasses other process models. Prototyping may be used in one spiral to resolve requirements uncertainties and hence reduce risk. This may be followed by a conventional waterfall development.
- **ADVANTAGES / PROS**
  - ✓ High amount of risk analysis hence, avoidance of Risk is enhanced.
  - ✓ Good for large and mission-critical projects.
  - ✓ Strong approval and documentation control.
  - ✓ Additional Functionality can be added at a later date.
  - ✓ Software is produced early in the software life cycle.
- **DISADVANTAGES / CONS**
  - ✓ Can be a costly model to use.
  - ✓ Risk analysis requires highly specific expertise.
  - ✓ Project's success is highly dependent on the risk analysis phase.
  - ✓ Doesn't work well for smaller projects.
- **APPLICATIONS**
  - ✓ When costs and risk evaluation is important
  - ✓ For medium to high-risk projects
  - ✓ Long-term project commitment unwise because of potential changes to economic priorities
  - ✓ Projects in which the users are unsure of their needs
  - ✓ Projects in which the requirements are complex.

SELECTING AN APPROPRIATE LIFE CYCLE MODEL FOR A PROJECT

- ❖ **Characteristics of the software to be developed:** If the software is simple data processing application an iterative waterfall model should be sufficient. An evolutionary model is a suitable model for object oriented development projects.
- ❖ **Characteristics of the development team:** If the development team is experienced in developing similar projects, then we can follow iterative waterfall model. If the team is entirely novice, then a prototyping model can be used.
- ❖ **Characteristics of the customer:** If the customer is not familiar with computers, then requirements are likely to change frequently. So prototyping model may be necessary to reduce the later change requests from customers.

3. SOFTWARE REQUIREMENT ANALYSIS & SPECIFICATION

- The goal of the requirements analysis phase is to clearly understand the customer requirements and to systematically organize the requirements into a specification document. The SRS document is the final outcome of the requirements analysis and specification phase. This phase is done by a system analyst.
- The three main types of problems in the requirements that the analyst needs to identify and resolve are
  i) **Ambiguity** – A requirement is ambiguous when several interpretations of that requirement are possible.
  ii) **Inconsistency** – Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
  iii) **Incompleteness** – An incomplete set of requirements is one in which some requirements have been overlooked. *(overlooked meaning = not considered)*

SOFTWARE REQUIREMENTS

IEEE defines requirement as

(i) A condition or capability needed by a user to solve a problem or achieve an objective.
(ii) A condition or a capability that must be met or possessed by a system to satisfy a contract standard, specification or other formally imposed document.

SOFTWARE REQUIREMENT SPECIFICATION

- IEEE defines SRS as a '*document that clearly and precisely describes each of the essential requirements [functions, performance, design constraints and quality attributes] of the software and the external interfaces*'.
- SRS describes what the proposed software should do without describing how the software will do it.
- Parts of a SRS document are
  i) Functional requirements of the system
  ii) Non- Functional requirements of the system
  iii) Goal of Implementation
- **Functional Requirements:** Those functionalities required by the users from the system or those functionalities that can be expressed in terms of functions. Functional requirement specify which output should be produced from the given inputs. They describe the relationship between the input and output of the system.

Example 1: Consider the case of a online calculator: The various functional requirements may be:
i) Addition
ii) Subtraction
iii) Multiplication
iv) Division etc.

Example 2: Consider the case of an ATM software: The various functional requirements are

i) Withdrawal
ii) Balance check
iii) Pin change
iv) Fund transfer
v) Print mini statement etc.

Example 3: Consider the case of a Library management software: The various functional requirements are

1. Register member
2. Register book
3. Issue book
4. Search a book
5. Return book
6. Reserve a book
7. Display available books
8. Display registered members
9. Display the list of borrowed books
10. Display the list of available books etc.

Example 4: Consider the case of a Music player software: The various functional requirements are

i) Play a song
ii) Stop playing song
iii) Pause playing song
iv) Shuffle the playlist
v) Fast forward
vi) Reduce volume level etc.

- **Non –Functional Requirements:** These are the various abilities / features possessed by a system or those characteristics of a system which cannot be expressed as functions.
  Eg: Scalability, maintainability, portability, reliability, availability, capacity, security, usability etc.
- **Goal of implementation:** This part of SRS, documents issues such as revisions to the system functionalities that may be required in future, new devices to be supported in the future, reusability issues etc.

NEED FOR SRS / WHY SRS IS IMPORTANT IN SOFTWARE DEVELOPMENT?

- Software SRS establishes the basic for agreement between the client and the supplier on what the software product will do.
- A SRS provides a reference for validation of the final product.
- A high-quality SRS is a prerequisite to high-quality software.
- A high-quality SRS reduces the development cost.

CHARACTERISTICS OF A GOOD SRS

Correct – Correctness of an SRS ensures that what is specified is done correctly.

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.

Note*:* *Correctness ensures that what is specified is done correctly, completeness ensures that everything is indeed specified.*

Concise – An SRS should be concise. That is it should specify only the relevant details.

Unambiguous - An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.

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.

Consistent - An SRS is consistent if there is no requirement that conflicts with another.

Ranked for importance and/or stability – All requirements are not equally important, hence each requirement is identified to make differences among other requirements.

Modifiable - The requirements of the user can change, hence requirements document should be created in such a manner that those changes can be modified easily, consistently maintaining the structure and style of the SRS.

Traceable - An SRS is traceable if the origin of each of 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.

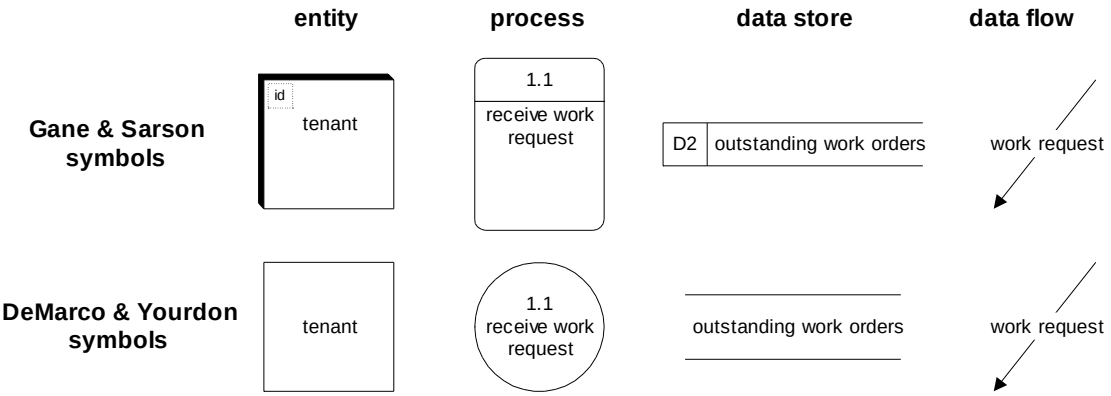
CATAGORIES OF USERS OF SRS DOCUMENT

- ✓ Users, customers and marketing personnel
- ✓ Software developers
- ✓ Test engineers
- ✓ User documentation writers
- ✓ Project managers
- ✓ Maintenance engineers

4. DFD,ER DIAGRAMS, DECISION TREE & DECISION TABLE

DATA FLOW DIAGRAMS (DFD)

- The DFD is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on that data, and the output generated by the system.
- DFD is used to represent the data transformations. That is it shows how input data is transformed into suitable output data.
- DFD is also termed as Bubble chart though a process is represented using bubble.
- Advantages of DFD are:
    i. DFDs are simple to understand and use
    ii. It uses a very few primitive symbols to represent the functions performed by a system and the data flow among these functions
- There are two different notations used in DFDs.
    i. Gane and Sarson notation
    ii. DeMarco and Yourdon notation



Entities:

- Those physical entities external to the software system such as people, departments, other companies, other systems or even a logical entity like bank account.
- Entities are called **sources** if they are external to the system and provide data to the system, and **sinks** if they are external to the system and receive information from the system

Processes:

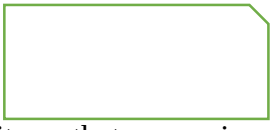
- A function / process is represented using a circle.
- A process must have at least one input and at least one output
- A process should be numbered.
- The name of the process should be specified in the bubble.

Data stores:

- Data store represents a logical file or a data base or any media which stores data. It can be online or "hard copy" .
- Data stores are labelled with a noun (e.g. the label "customer" indicates that information about customers is kept in that data store)
- Data is stored whenever there are more than one process that needs it and these processes don't always run one after the other (if the data is ever needed in the future it must be stored)

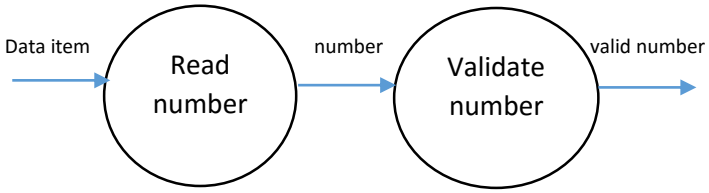
Data flows:

- It must originate from and/or lead to a process (this means that entities and data stores cannot communicate with anything except processes –remember that it takes a process to make the data flow)
- It can go from process to process, but that does imply that no data is stored at that point
- It can have one arrowhead indicating the direction in which the data is flowing
- It can have 2 arrowheads when a process is altering (updating) existing records in a data store
- In De Marco and Yourdon notation an additional symbol is mentioned for representing hardcopy output.

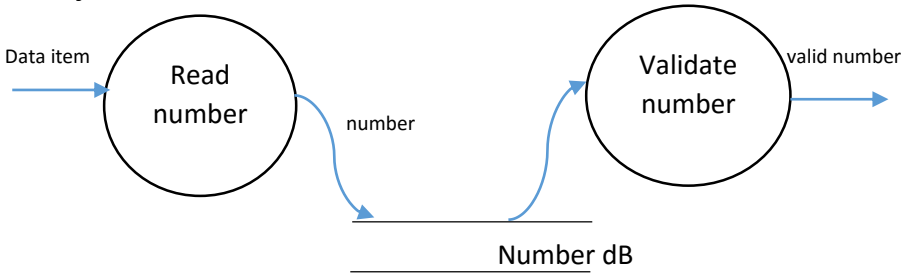


- A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and lists all data items that appear in a DFD model.

- **Synchronous and Asynchronous operations:**
  - If two processes are directly connected by a dataflow arrow, then they are synchronous. This means that they operate at the same speed.



- If two processes are connected through a data store, then the speed of operation is independent. They are asynchronous.



- **Short comings of DFD / Disadvantages:**
  - **i.** DFDs leave ample scope to be imprecise
  - **ii.** Control aspects are not defined by a DFD
  - **iii.** Though decomposition is possible, same problem has several alternative DFD representations.

CONTEXT DIAGRAM / LEVEL 0 DFD

Context diagram is the highest level of data flow representation of a system. It represents the entire system as a single bubble. The bubble in context diagram is annotated with the name of the software system being developed.

- ✓ Context diagram establishes the context in which the system operates. That is who the users are, what data do they input to the system and what data do they received from the system.
- ✓ All entities should be represented in the context diagram. No entities are represented in any other levels of DFD.

DRAWING DFDs

- ✓ Except for the context DFD, each **DFD** represents the breakdown of one process.

  For example, in the hierarchy on the next page, the level 1 DFD that represents the breakdown of process 1.2 will contain processes 1.2.1, 1.2.2 and 1.2.3. But it will not contain 1.2, nor any other process.

- ✓ **Context level** diagrams show all external entities. They do not show any data stores. The context diagram always has only one process labeled 0.
- ✓ When you draw a **level 0** diagram, follow these rules:
  - include all entities in the context diagram
  - show any data store that are shared by the processes in the level 0 diagram
- ✓ When you draw a **level 1 or 2 etc**. diagram, follow these rules:
    - include all entities and data stores that are directly connected by data flow to the one process you are breaking down
    - show all other data stores that are shared by the processes in this breakdown (these data stores are "internal" to this diagram and will not appear in higher level diagrams, but will appear in lower level diagrams)

That last statement is often confusing. Here is another explanation using the hierarchy on If a data store is used only by processes 3.2.1 and 3.2.3, then it will appear only in the level 2 diagram that includes processes 3.2.1, 3.2.2 and 3.2.3. It will not appear in the diagram that shows processes 3.1 and 3.2 because it is internal to process 3.2.

✓ A DFD that contains processes that are not further broken down is called a **primitive** DFD.

GUIDELINES FOR DRAWING DFD

- Naming conventions:
  - Processes: strong verbs
  - Data flows: nouns
  - data stores: nouns
  - external entities: nouns
- No more than 7 - 9 processes in each DFD.
- Dataflow must begin, end, or both begin & end with a process.
- Dataflow must not be split.
- A process is not an analogy of a decision in a systems or programming flowchart. Hence, a dataflow should not be a control signal. Control signals are modelled separately as control flows.
- Loops are not allowed.
- A dataflow cannot be an input signal. If such a signal is necessary, then it must be a part of the description of the process, and such process must be so labelled. Input signals as well as their effect on the behaviour of the system are incorporated in the behavioural model (say, state transition graphs) of the information system.
- Decisions and iterative controls are part of process description rather than dataflow.
- If an external entity appears more than once on the same DFD, then a diagonal line is added to the north-west corner of the rectangle (representing such entity).
- Updates to data store are represented in the textbook as double-ended arrows. This is not, however, a universal convention. I would rather you did not use this convention since it can be confusing. Writing to a data store implies that you have read such data store (you cannot write without reading). Therefore, data store updates should be denoted by a single-ended arrow from the updating process to the updated data store.
- Data flows that carry a whole record between a data store and a process is not labelled in the textbook since there is no ambiguity. This is also not a universal convention. I would rather you labelled such data flows explicitly.
- **Conservation Principles:**

  Data stores & Data flows: Data stores cannot create (or destroy) any data. What comes out of a data store therefore must first have got into a data store through a process.
  Processes: Processes cannot create data out of thin air. Processes can only manipulate data they have received from data flows. Data outflows from processes therefore must be derivable from the data inflows into such processes.

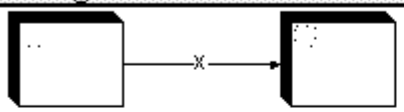
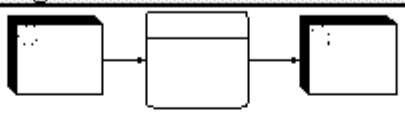
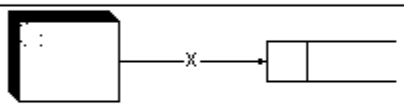
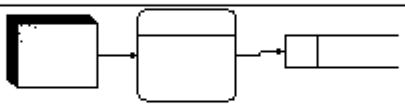
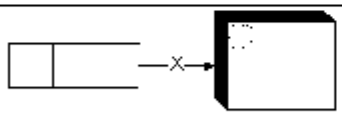
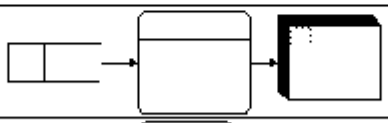
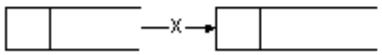
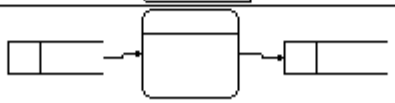
- **Levelling Conventions:**

  ➢ Numbering: The system under study in the context diagram is given number `0'. The processes in the top level DFD are labelled consecutively by natural numbers beginning with 1. When a process is exploded in a lower level DFD, the processes in such lower level DFD are consecutively numbered following the label of such parent process ending with a period or full-stop (for example 1.2, 1.2.3, etc.).

- Balancing: The set of DFDs pertaining to a system must be balanced in the sense that corresponding to each dataflow beginning or ending at a process there should be an identical dataflow in the exploded DFD.
- Data stores: Data stores may be local to a specific level in the set of DFDs. A data store is used only if it is referenced by more than one process.
- External entities: Lower level DFDs cannot introduce new external entities. The context diagram must therefore show all external entities with which the system under study interacts. In order not to clutter higher level DFDs, detailed interactions of processes with external entities are often shown in lower level DFDs but not in the higher level ones.

COMMONLY MADE MISTAKES IN DFDS

Table 2: Common data flow diagramming mistakes

Wrong	Right	Description
		A source or a sink cannot provide data to another source or sink without some processing occurring.
		Data cannot move directly from a source to a data store without being processed.
		Data cannot move directly from a data store to a sink without being processed.
		Data cannot move directly from one data store to another without being processed.

Source: Adapted from Figure 9.9, p. 360 in Whitten, J. L.; Bentley, L. D.; Barlow, V. M. (1994). Systems Analysis and Design Methods (Third Edition). Burr Ridge, IL: Irwin.

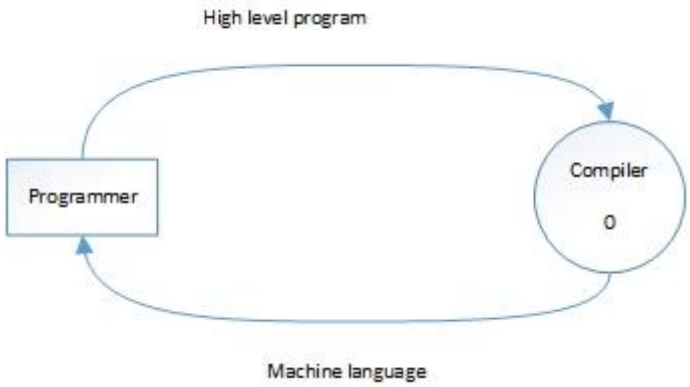
Diagramming mistakes: Black holes, grey holes, and miracles

A second class of DFD mistakes arise when the outputs from one processing step do not match its inputs. It is not hard to list situations in which this might occur:

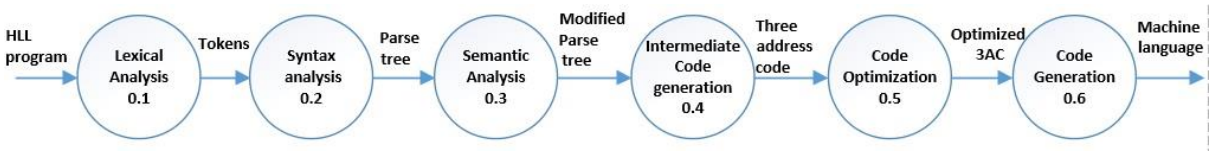
- A processing step may have input flows but no output flows. This situation is sometimes called a *black hole* .
- A processing step may have output flows but now input flows. This situation is sometimes called a *miracle*.
- A processing step may have outputs that are greater than the sum of its inputs - e.g., its inputs could not produce the output shown. This situation is sometimes referred to as a *grey hole*.

DFD OF COMPILER

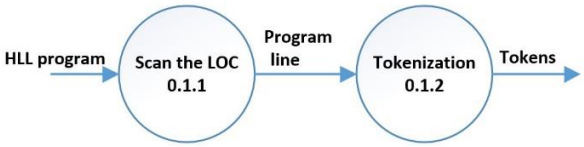
LEVEL 0 DFD: CONTEXT DIAGRAM OF COMPILER



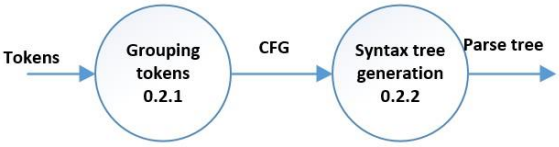
LEVEL 1 DFD: COMPILER



LEVEL 2.1 DFD: LEXICAL ANALYSIS



LEVEL 2.2 DFD: SYNTAX ANALYSIS



ENTITY –RELATIONSHIP (ER) DIAGRAMS

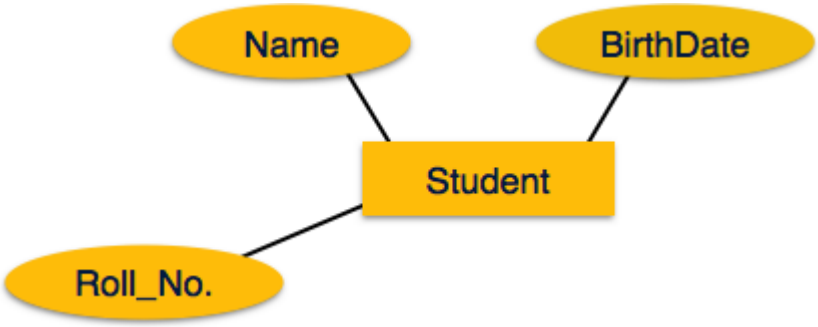
ER diagram represents the various entities in the system and the relationship exist between those entities. Any object, for example, entities, attributes of an entity, relationship sets, and attributes of relationship sets, can be represented with the help of an ER diagram.

Entity

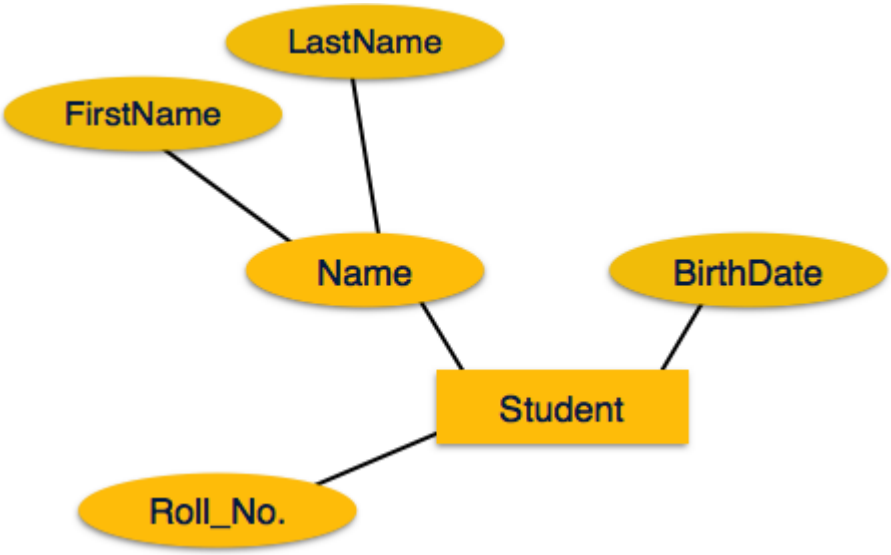
Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.

Attributes

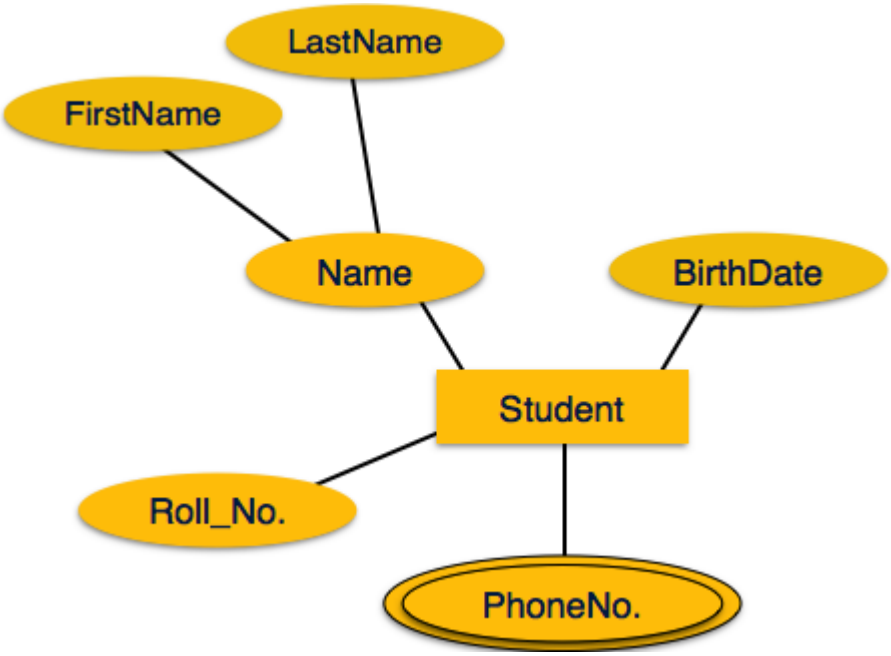
Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).



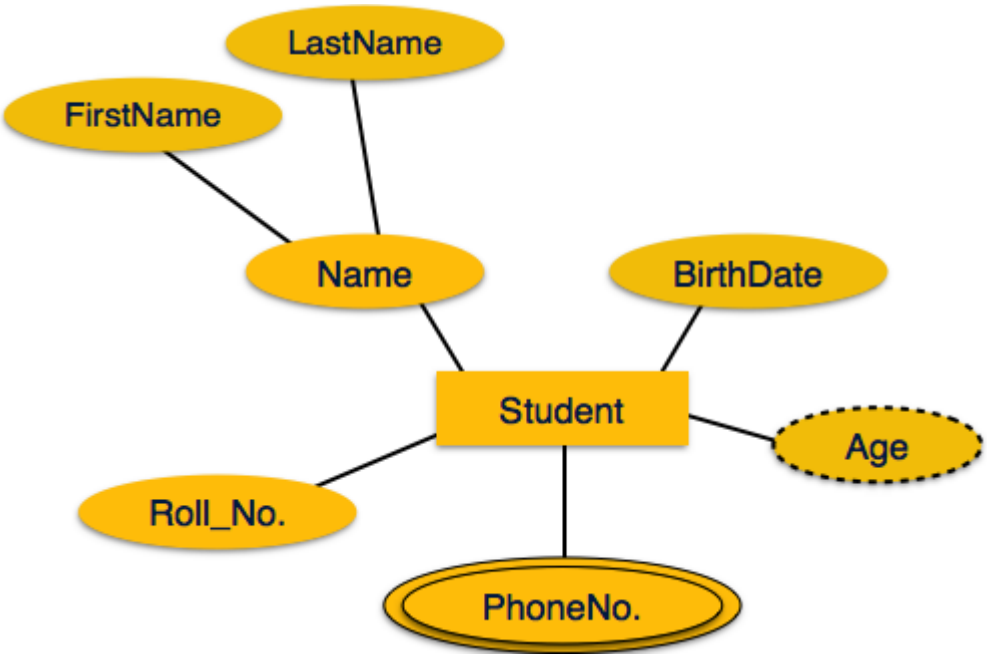
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.



Multivalued attributes are depicted by double ellipse.



Derived attributes are depicted by dashed ellipse.



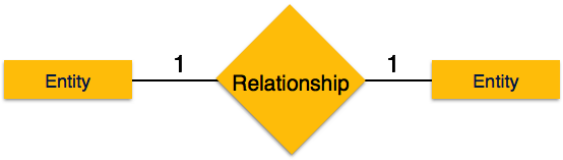
Relationship

Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

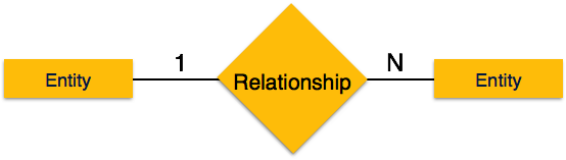
Binary Relationship and Cardinality

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

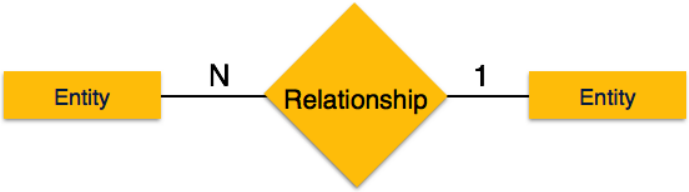
- **One-to-one** − When only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.



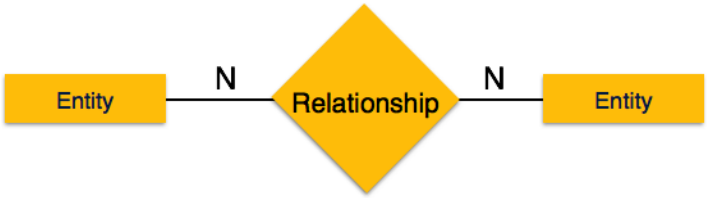
- **One-to-many** − When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.



- **Many-to-one** − When more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.



- **Many-to-many** − The following image reflects that more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts many-to-many relationship.



STEPS FOR DEVELOPING ER DIAGRAM

1. Identify the entities associated with the system.
2. Specify the attributes of each entity.
3. Identify the relationship that exist between the identified entities.
4. Specify the cardinality.

Example 1: ER DIAGRAM FOR ATM TRANSACTION

Entities: ATM, Card holder/ User, Bank

Attributes: **ATM –** ATM-id, Location, Bank name etc.

Card holder – Name, Card_number, Account no:, Bank name, Address etc.

Bank – Bank_name, Branch IFS Code, Location

Relationships:

- Card holder uses ATM by swiping the card / login to the ATM. After login the user performs various transactions. An ATM can be used by many card holders. Also a card holder can use any ATM provided by different bank groups. So the relationship cardinality exist between ATM and card holder is *many to many*.
- ATMs are managed by a bank. A particular ATM is managed by a single Bank group. But a single Bank group can manage multiple ATMs in different locations

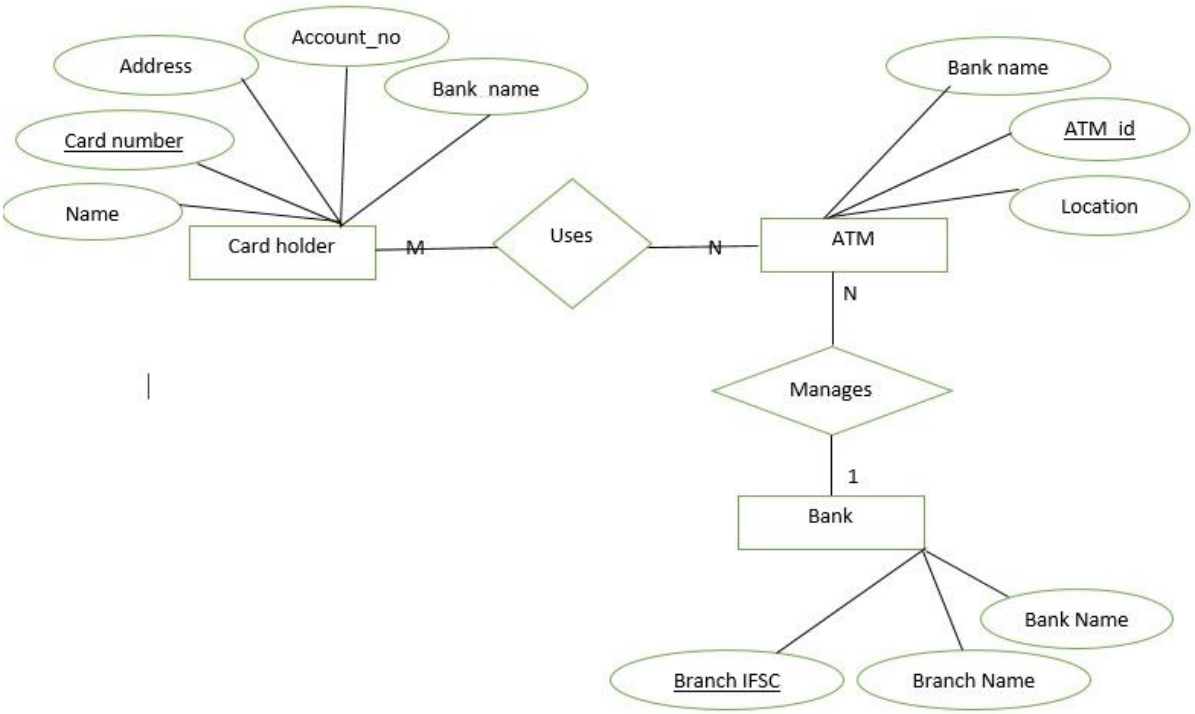
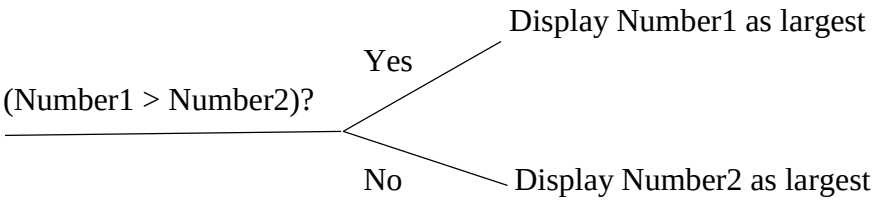


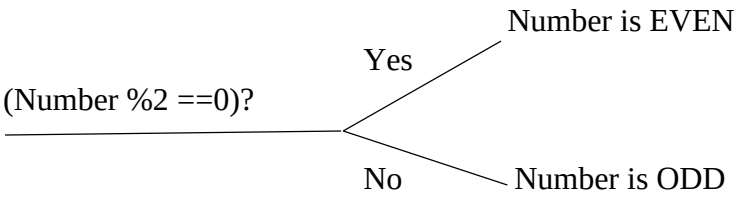
Figure: ER diagram of ATM

DECISION TREE

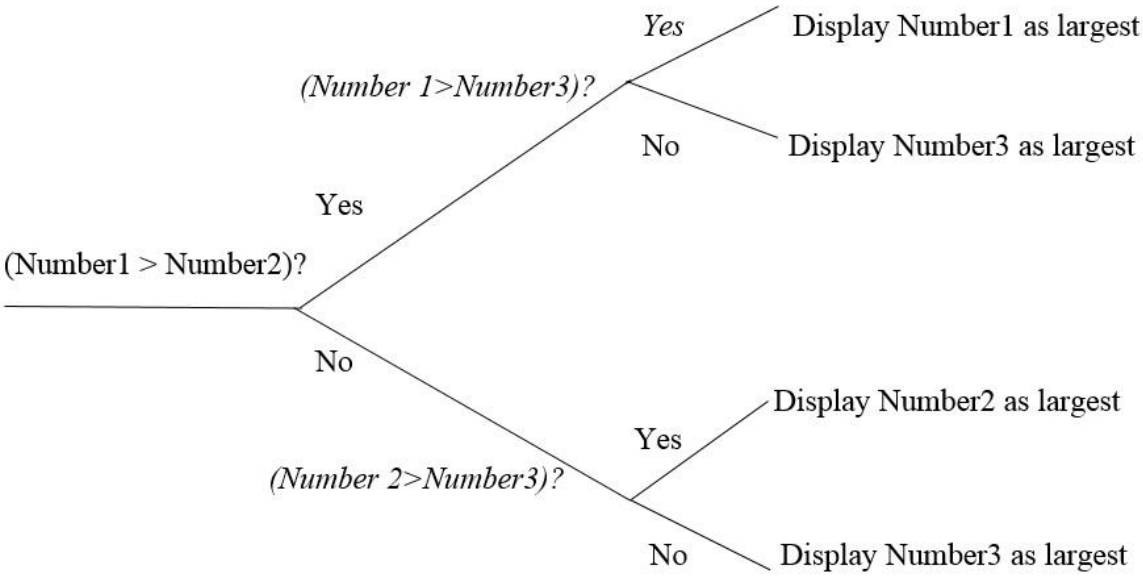
- A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken.
- The edges of the decision tree represent condition and the leaf nodes represent the actions to be performed depending on the outcome of testing the conditions.
- **Example 1:** Decision tree to represent the logic of '*Largest of two integers'*



- **Example 2:** Decision tree to represent the logic of '*Checking odd or even'*



- **Example 3:** Decision tree to represent the logic of '*Largest of three integers'*



DECISION TABLE

- A decision table shows the decision – making logic and the corresponding actions taken in a tabular or a matrix form.

- A decision table is composed of rows and columns, which contain four separate quadrants as shown below.

Conditions	Condition alternatives
Actions	Action entries

1. **Conditions:** Contains a list of all possible conditions pertaining to a problem.
2. **Condition alternatives / Rules:** Contains the condition rules (set of possible values for each condition) of alternatives.
3. **Actions:** Contains actions, which can be a procedure or operation to be performed.
4. **Action entries:** Contains the Action rules, which specify the actions to be performed on the basis of the set of condition alternatives corresponding to that action entry.

- **Example 1: Decision table for 'Largest of 2 numbers'.**

		Rules	
Conditions	**Number1> Number2**	**T**	**F**
Actions	**Display Number1 is largest**	**X**	
	Display Number2 is largest		**X**

- **Example 2: Decision table for 'Printer trouble shooting'.**

		RULES							
CONDITIONS	**Printer does not print**	T	T	T	T	F	F	F	F
	A red light is flashing	T	T	F	F	T	T	F	F
	Printer is unrecognized	T	F	T	F	T	F	T	F
ACTIONS	**Check the power cable**	X		X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check / Replace ink	X	X			X	X		
	Check for paper jam	X	X		X				

- **Example 3: Decision table for 'Mobile phone trouble shooting'.**

		RULES			
CONDITIONS	**Phone gets hang**	T	T	F	F
	Fast discharging / Battery drainage	T	F	T	F
ACTIONS	**Restart the phone**	X	X		
	Close running applications	X	X	X	
	Recharge phone	X		X	
	Clear memory	X	X		
	Reduce brightness/ Switch to power save mode	X		X	

DECISION TREE VS DECISION TABLE

- **Readability:** Decision trees are easier to understand when the number of conditions are small. But a decision table causes the analyst to look at every possible combination of conditions which the analyst might otherwise omit.
- **Explicit representation of the order of decision making:** Contrast to decision trees, order of decision making is abstracted out in decision tables. Decision trees are more useful to represent multilevel decision making hierarchically, whereas decision table can only represent a single decision to select the appropriate action for execution.
- **Representing very complex decision logic:** When number of conditions are large, decision tree become very complex to understand. So we prefer decision tables when conditions and actions increases.

5. FORMAL SPECIFICATION TECHNIQUES

Formal technique

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by the specification language.

Formal specification language

A formal specification language consists of two sets syn and sem, and a relation sat between them. The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn, and model of the system sem, if sat (syn, sem), as shown in figure below, then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

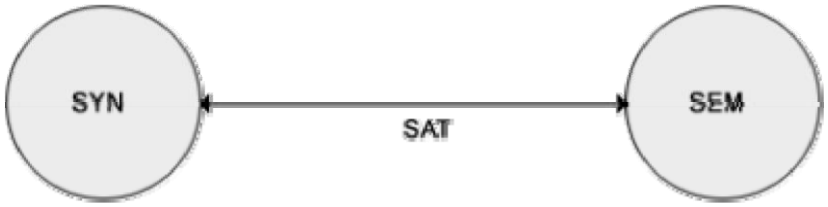


Fig. 3.6: sat (syn, sem)

- **Syntactic Domains**

The syntactic domain of a formal specification language consists of an alphabet of symbols and set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.

- **Semantic Domains**

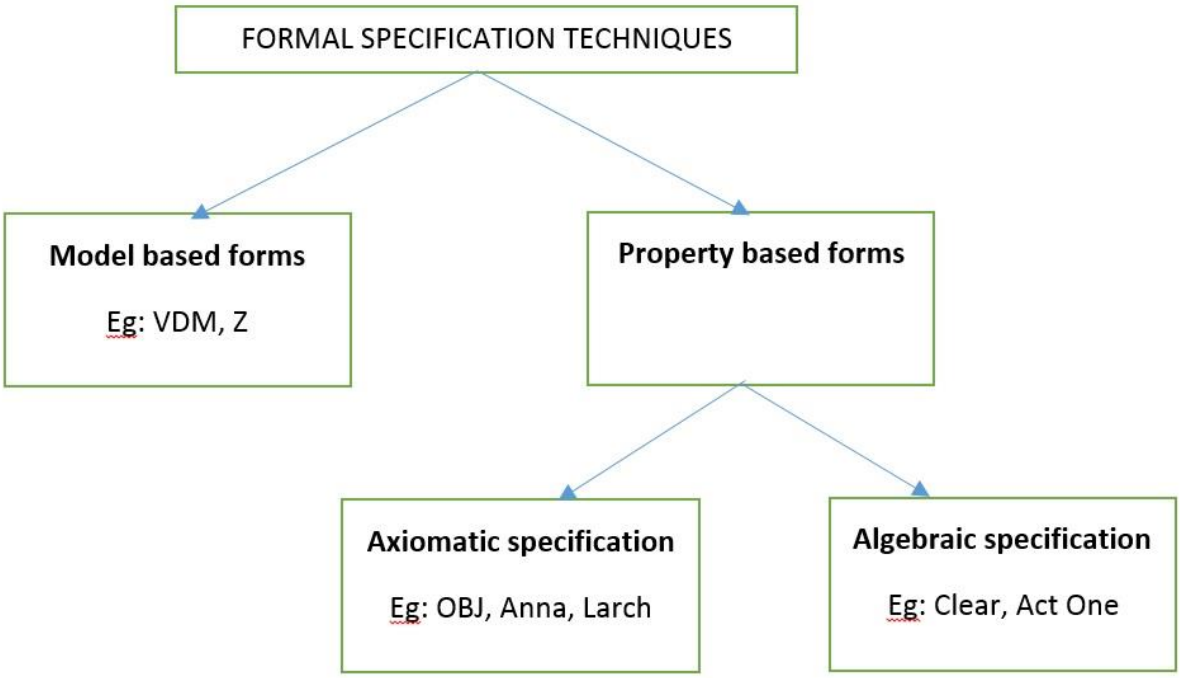
Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify

functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronization trees, partial orders, state machines, etc.

- **Satisfaction Relation**

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as semantic abstraction function. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined: those that preserve a system's behaviour and those that preserve a system's structure.

FORMAL SPECIFICATION CATEGORIES



MODEL ORIENTED vs PROPERTY ORIENTED

- Formal methods are usually classified into two broad categories – model – oriented and property – oriented approaches.
- In a model-oriented style, one defines a system's behaviour directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. Examples of popular model-oriented specification techniques are Z, CSP, CCS, etc.
- In the property-oriented style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Examples of property-oriented specification styles are axiomatic specification and algebraic specification.
- Model-oriented approaches are more suited to use in later phases of life cycle because here even minor changes to a specification may lead to drastic changes to the entire specification. They do not support logical conjunctions (AND) and disjunctions (OR).

- Property-oriented approaches are suitable for requirements specification because they can be easily changed. They specify a system as a conjunction of axioms and you can easily replace one axiom with another one.

MERITS OF FORMAL SPECIFICATION TECHNIQUES

1. **Formal specifications encourage rigour:** Often, the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behaviour that are not obvious in an informal specification.
2. **More precise and mathematically sound:** Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties of a specification and to rigorously prove that an implementation satisfies its specifications.
3. **Reduces ambiguity in specification:** Formal methods have well-defined semantics. Therefore, ambiguity in specifications is automatically avoided when one formally specifies a system.
4. **Facilitates specification analysis automation:** The mathematical basis of the formal methods facilitates automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
5. **Formal specifications can be executed to obtain immediate feedback on the features of the specified system:** This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a "toy" working model of a system that can provide immediate feedback on the behaviour of the specified system, and is especially useful in checking the completeness of specifications.

LIMITATIONS OF FORMAL SPECIFICATION TECHNIQUES

6. Formal methods are difficult to learn and use.
7. The basic incompleteness results of first-order logic suggest that it is impossible to check absolute correctness of systems using theorem proving techniques.
8. Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

AXIOMATIC SPECIFICATION

- In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully. Thus, the post conditions are essentially constraints on the results produced for the function execution to be considered successful.

STEPS FOR DEVELOPING AXIOMATIC SPECIFICATION

1) Establish the range of input values over which the function should behave correctly. Also find out other constraints on the input parameters and write it in the form of a predicate.
2) Specify a predicate defining the conditions which must hold on the output of the function if it behaved properly.
3) Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
4) Combine all of the above into pre and post conditions of the function.

Example 1: Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

f (x) : real

x : real

pre : $x \in R$

post : $\{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2 * x)\}$

Example 2: Specify the pre- and post-conditions of a function that takes a real number as argument and returns square of the input number, if the input is even number, or else returns square root of the value.

f (x) : real

x : real

pre : $x \in R$

post : $\{(x\%2==0) \wedge (f(x) = x^2)\} \vee \{(x\%2!=0) \wedge (f(x) = x^{1/2})\}$

Example 3: - Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

search(X : IntArray, key : Integer) : Integer pre : $\exists$ i $\in$

[Xfirst….Xlast], X[i] = key

post : $\{(X'[search(X, key)] = key) \wedge (X = X')\}$

Here the convention followed is: If a function changes any of its input parameters and if that parameter is named X, then it is referred to as X′ after the function completes execution.

ALGEBRAIC SPECIFICATION

- In the algebraic specification technique an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980, 1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

- Algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations; {I, +, -, *, /}. In contrast, alphabetic strings together with operations of concatenation and length {A, I, con, len}, is not a homogeneous algebra, since the range of the length operation is the set of integers.
- Each set of symbols in the algebra, in turn, is called a *sort* of the algebra. To define a heterogeneous algebra, we first need to specify its signature, the involved operations, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using equations.
- An algebraic specification is usually presented in four sections.
  1) **Types section:-** In this section, the sorts (or the data types) being used is specified.
  2) **Exceptions section:-** This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification. For example, in a queue, possible exceptions are novalue (empty queue), underflow (removal from an empty queue), etc.
  3) **Syntax section:-** This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the signature of the operator. For example, the add operation of two integers is represented as *sum= add(int a, int b)*
  4) **Equations section:-** This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions. For example, a rewrite rule to identify an empty queue may be written as: *isempty(create()) = true*
- Categories of operators in algebraic specification are as follows:

1. **Basic construction operators.** These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, ‘create’ and ‘append’ are basic construction operators for a FIFO *queue*.
2. **Extra construction operators.** These are the construction operators other than the basic construction operators. For example, the operator ‘remove’ is an extra construction operator for a FIFO queue because even without using ‘remove’, it is possible to generate all values of the type being specified.
3. **Basic inspection operators.** These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let S be the set of operators whose range is not the data type being specified. The set of the basic operators S1 is a subset of S, such that each operator from S-S1 can be expressed in terms of the operators from S1. For example, ‘isempty’ is a basic inspection operator because it does not modify the FIFO queue type.
4. **Extra inspection operators.** These are the inspection operators that are not basic inspectors.

Example:- Let us specify a FIFO queue supporting the operations *create*, *append*, *remove*, *first*, and *isempty* where the operations have their usual meaning.

Types:
defines queue
uses boolean, integer

Exceptions: underflow, novalue

Syntax:

1. *create : φ → queue*
2. *append : queue x element → queue*
3. *remove : queue → queue + {underflow}*
4. *first : queue → element + {novalue}*
5. *isempty : queue → boolean*

Equations:

1. *isempty(create()) = true*
2. *isempty((append(q,e)) = false*
3. *first(create()) = novalue*
4. *first(append(q,e)) = is isempty(q) then e else first(q)*
5. *remove(create()) = underflow*
6. *remove(append(q,e)) = if isempty(q) then create() else append(remove(q),e)*

In this example, there are two basic constructors (*create* and *append*), one extra construction operator (*remove*) and two basic inspectors (*first* and *empty*). Therefore, there are 2 x (1+2) + 0 = 6 equations.

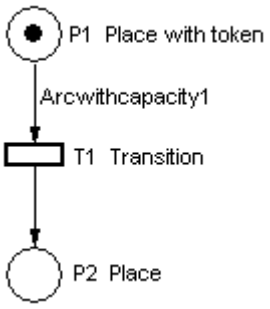
- **Properties of algebraic specification:** Three important properties that every algebraic specification should possess are:
  i) **Completeness:** This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.
  ii) **Finite termination property:** This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.
  iii) **Unique termination property:** This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked: Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same number? Checking the unique termination property is a very difficult problem.

- **Advantages and disadvantages of algebraic specifications**

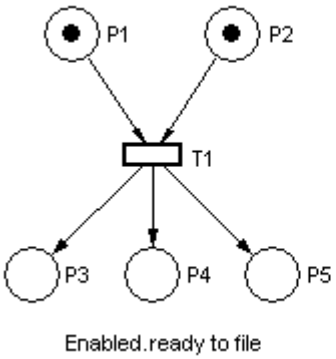
Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be studied. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to interchange with typical programming languages. Also, algebraic specifications are hard to understand.

PETRI NETS

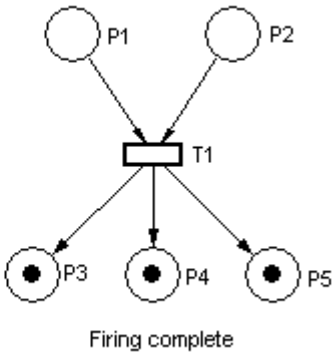
- Petri Nets were developed originally by Carl Adam Petri [Pet62], and were the subject of his dissertation in 1962. Since then, Petri Nets and their concepts have been extended and developed, and applied in a variety of areas: Office automation, work-flows, flexible manufacturing, programming languages, protocols and networks, hardware structures, real-time systems, performance evaluation, operations research, embedded systems, defence systems, telecommunications, Internet, e-commerce and trading, railway networks, biological systems.
- Similar to state – transition diagram.
- A Petri Net is a collection of directed arcs connecting places and transitions. Places may hold tokens. The state or marking of a net is its assignment of tokens to places. Here is a simple net containing all components of a Petri Net:



- Arcs have capacity 1 by default; if other than 1, the capacity is marked on the arc. Places have infinite capacity by default, and transitions have no capacity, and cannot store tokens at all. With the rule that arcs can only connect places to transitions and vice versa.
- A transition is enabled when the number of tokens in each of its input places is at least equal to the arc weight going from the place to the transition. An enabled transition may fire at any time. When fired, the tokens in the input places are moved to output places, according to arc weights and place capacities. This results in a new marking of the net, a state description of all places.

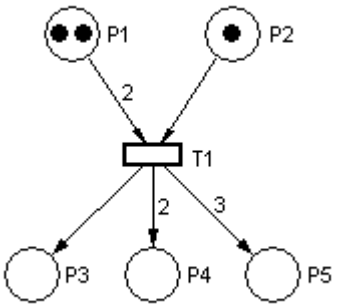


Enabled.ready to file

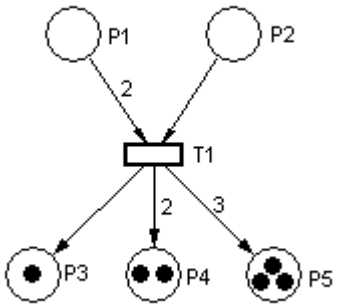


Firing complete

- When arcs have different weights, we have what might at first seem confusing behaviour. Here is a similar net, ready to fire and it is after firing.



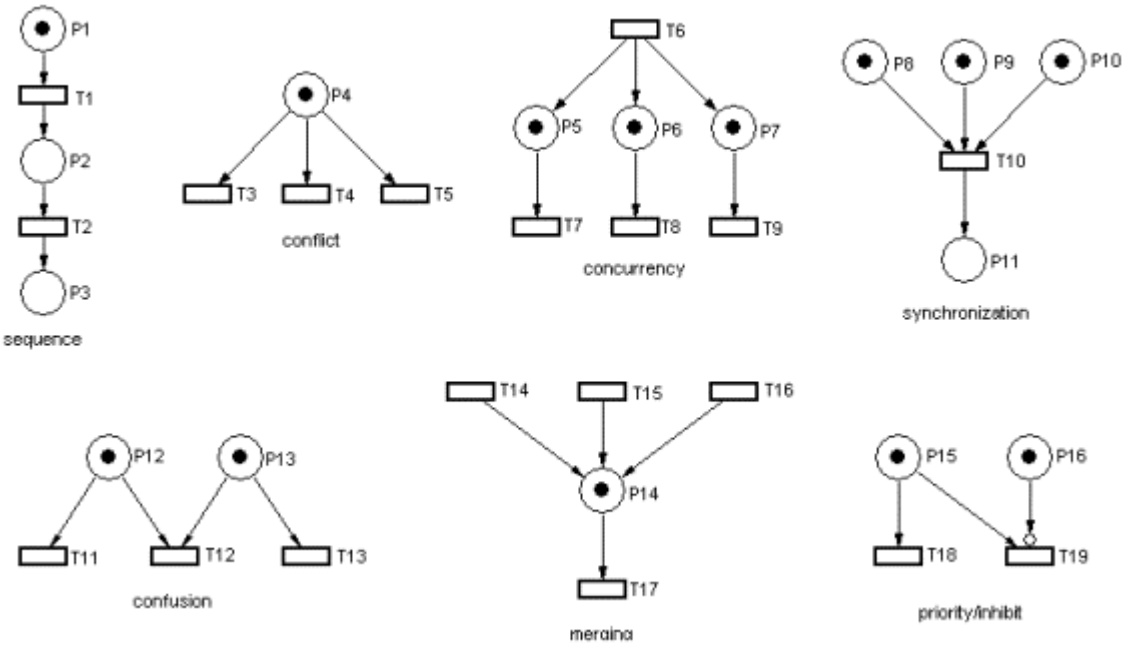
Before firing



After firing

- When a transition fires, it takes the tokens that enabled it from the input places; it then distributes tokens to output places according to arc weights. If the arc weights are all the same, it appears that tokens are moved across the transition. If they differ, however, it appears that tokens may disappear or be created. That, in fact, is what happens; think of the transition as removing its enabling tokens and producing output tokens according to arc weight

- Here is a collection of primitive structures that occur in real systems, and thus we find in Petri Nets.



- Sequence is obvious - several things happen in order. Conflict is not so obvious. The token in P4 enables three transitions; but when one of them fires, the token is removed, leaving the remaining two

disabled. Unless we can control the timing of firing, we don't know how this net is resolved. Concurrency, again, is obvious; many systems operate with concurrent activities, and this models it well. Synchronization is also modelled well using Petri Nets; when the processes leading into P8, P9 and P10 are finished, all three are synchronized by starting P11.

- Confusion is another not so obvious construct. It is a combination of conflict and concurrency. P12 enables both T11 and T12, but if T11 fires, T12 is no longer enabled.
- Merging is not quite the same as synchronization, since there is nothing requiring that the three transitions fire at the same time, or that all three fire before T17; this simply merges three parallel processes. The priority/inhibit construct uses the inhibit arc to control T19; as long as P16 has a token, T19 cannot fire.

6. DESIGN HEURISTICS

MODULARIZATION

- Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
- Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

LAYERED DESIGN

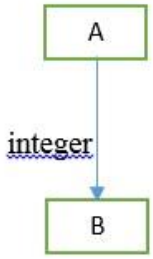
- In layered design methodology, modules are arranged in a hierarchy of layers.
- A module can invoke functions of the modules in the layer immediately below it.
- A layered design provides control abstraction and is easily understandable.
- It is easier to perform debugging (correcting errors) in a layered design.

COUPLING

- The degree of interdependence between two modules or the degree of interaction between two modules.
- For a good software design, coupling should be as minimum as possible.
- Coupling is basically classified into two:
    1. **Tightly coupled / High coupled:** Two modules are said to be tightly coupled if they share large amount of data.
    2. **Loosely coupled / Low coupled:** Two modules are said to be loosely coupled if they do not interact with each other or they share less amount of data.

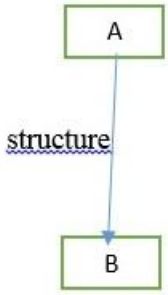
TYPES OF COUPLING

- **Data coupling:** Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two. Elementary data items are like an integer, a float, a character etc.



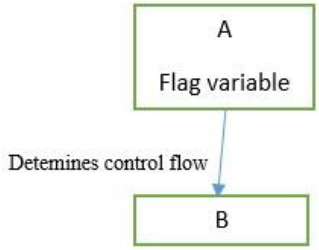
Consider the modules A and B. These modules are related with each other in such a way that they share an integer value which is an elementary data item. So the modules A and B are said to be data coupled.

- **Stamp coupling:** Two modules are stamp coupled, if they communicate using a composite data item. Composite data items are like Structure in C or Record in Pascal or a List in Python etc.



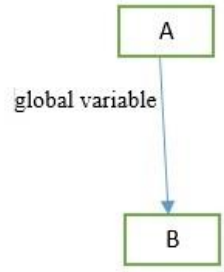
Consider the modules A and B. These modules are related with each other in such a way that they share a structure which is a composite data item. So the modules A and B are said to be stamp coupled.

- **Control coupling:** Two modules are said to be control coupled, if data from one module is used to direct the order of instruction execution in another.



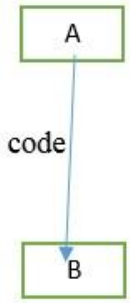
Consider the modules A and B. These modules are related with each other in such a way that the flag variable in module A determines the flow of control of module B. So the modules A and B are said to be control coupled.

- **Common coupling:** Two modules are common coupled, if they share some global data items. Global data items are those variables we use in a program globally.



Consider the modules A and B. These modules are related with each other in such a way that they share a globally declared variable. So the modules A and B are said to be common coupled though the global variable is common to all modules.

❖ **Content coupling:** Two modules are content coupled, if they share code.



Consider the modules A and B. These modules are related with each other in such a way that they share program code itself. So the modules A and B are said to be content coupled.

Degree of coupling:

Data coupling < Stamp coupling < Control coupling < Common coupling < Content coupling

COHESION

- Cohesion of a module is the degree to which the different functions of the module cooperate to work towards a single objective.
- Cohesion is termed as the measure of the functional strength of a module.
- Cohesion is broadly categorized into two:
    1. **High cohesion / Good cohesion:** When the functions of the module cooperate with each other for performing a single objective, then the module has good cohesion.
    2. **Low cohesion / Poor cohesion:** If the functions of the module do very different things and do not cooperate with each other to perform a single piece of work, then the module has very poor cohesion.

TYPES OF COHESION:

Coincidental cohesion: A module is said to have coincidental cohesion, if the functions in the module perform entirely different tasks or unrelated operations.

Module name : Main
Functions:
Add()
Sqrt()
Read()
Display()
Search()
Push()
Pop()

For example: In the above module named Main, the various functions are unrelated to each other and they operate on entirely different data. So the module is said to be in coincidental cohesion.

Logical cohesion: A module is said to have logical cohesion, if the functions in the module performs similar operations.

Module name : Stack **Functions:** Push() Pop()	**Module name :** IO **Functions:** Read() Display()

For example: In the above module named Stack, the functions push() and pop() are logically similar operations whereas the other module named IO performs the input/output functions like Read() and Display() which are similar operations. So the both modules are in logical cohesion.

Temporal cohesion: A module is said to have temporal cohesion, if the functions in the module are executed in the same time span.

Module name : System initialization **Functions:** Manage_memory() Manage_devices() Load_OS()

For example: In The above system initialization modules, the various activities like managing memory and other system devices, loading the operating system(OS) etc. should be performed during same time span. So the above module is in temporal cohesion.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions in the module are executed one after another, though these functions performs entirely different tasks and operate on very different data.

Module name : Order processing **Functions:** login() place_order() check_order() print_bill() update_stock() logout()

For example: Consider the activities associated with order processing in a online shopping website warehouse. The functions login(), place_order(), check_order(), print_bill(), update_stock() and logout() all do different things and operate on different data/ databases. However they are normally executed one after another in a typical order processing.

Communicational cohesion: A module is said to possess communicational cohesion, if all functions of the module refer to or update the same data structure or database.

Module name : Student
Functions:
Admit_student()
Enter_marks()
Print_gradesheet()
Sort_names()

For example: In the above module named student, the various functions like Admit_student(), Enter_marks(), Print_gradesheet(), Sort_names etc. are updating the same database which holds the student records.

Sequential cohesion: A module is said to possess sequential cohesion, if all the functions of the module are executed one after another in a sequence, and the output of one function is input to the next in the sequence.

Module name : SDLC
Functions:
Requirement_gathering()
SRSdocument_generation()
Designdocument_generation()
Testdocument_generation()

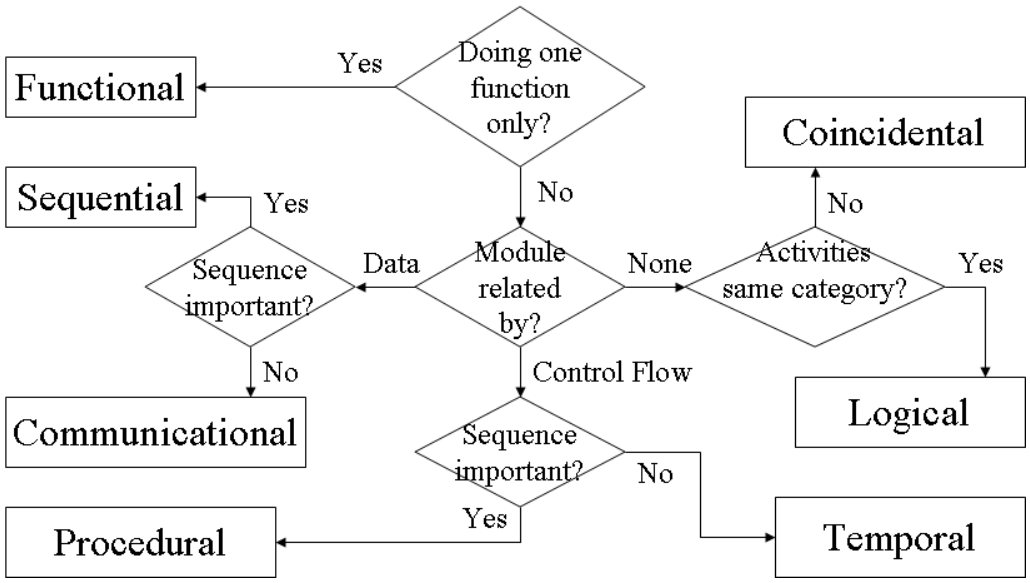
For example: Consider the various documenting activities associated with software development life cycle. The various documents like SRS, Design, Test are generated in a sequence such that the output of one phase is the input to the next in the sequence.

Functional cohesion: A module is said to possess functional cohesion, if different functions of the module cooperate to complete a single task.

Module name : Calculate Attendance Percentage
Functions:
compute_presentdays()
compute_dutydays()
computetotaldays()

For example: In the above module named *Calculate Attendance Percentage* the various functions like compute_presentdays(), compute_dutydays(), computetotaldays() etc. work together to calculate the attendance percentage.

COHESION CHART:



FUNCTIONAL INDEPENDENCE

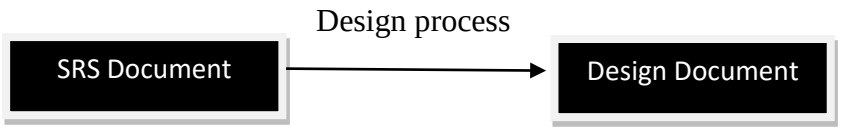
- A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.
- Advantages of functional independence are:
    1. **Error isolation:** When an error exists in a module, functional independence reduces the chances of propagating the error to the other modules.
    2. **Scope of reuse:** It will be easier to reuse a module which is functionally independent.
    3. **Understandability:** When modules are functionally independent, complexity of the design is greatly reduced and hence provides better understandability.

A good software design should possess functionally independent modules. That is the various modules in the design should has low coupling and high cohesion.

7. DESIGN METHODOLOGIES

SOFTWARE DESIGN

- During the design phase, the design document is produced based on customer requirements as documented in the SRS document.
- The activities carried out during the design phase (design process) transform the SRS document into the design document.
- The design document produced at the end of the design phase should be implementable using a programming language in the coding phase.



OUTCOME OF A DESIGN PROCESS

- **Different modules required:** The different modules in the solution should be identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should perform a well-defined task and is named according to the task it performs.
- **Data structures of the individual modules:** Suitable data structures for the data to be stored in a module needs to be properly designed and documented.
- **Control relationships among modules:** A control relationship between two modules essentially arises due to function calls between the two modules. The control relationships existing among various modules should be identified in the design document.
- **Interfaces among different modules:** The interfaces between two modules identifies the exact data items exchanged between the two modules when one module invokes a function of the other module.
- **Algorithms required to implement individual modules:** Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules should be designed and documented by considering accuracy of result, space and time complexity.

CHARACTERSITICS OF A GOOD SOFTWARE DESIGN

1. **Correctness:** A good design should correctly implement all the functionalities of the system.
2. **Understandability:** A good design should be easily understandable, otherwise it will be difficult to implement and maintain it. A design solution is understandable, if it is modular and the modules are arranged in layers. An understandable design solution should have the following features.
   - It should contain consistent and meaningful names to various design components.
   - It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
3. **Maintainability:** A good design solution should be easy to change because requirement change request arises from customer even after product release.
4. **Efficiency:** A good design solution should adequately address resource, time and cost optimization issues.
5. **Cost:** A good design solution should reduce costs in later phases of software development.

DESIGN HEURISTICS

TOP DOWN DESIGN

- ✓ A system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their own set of sub-system and components and creates hierarchical structure in the system.

- ✓ Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.
- ✓ Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.
- ✓ Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

BOTTOM UP DESIGN

- ✓ The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- ✓ Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- ✓ Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

MODULE

- ✓ It is a logically separable part of a program
- ✓ It is a program unit that is discrete and identifiable with respect to compiling and loading
- ✓ It can be a function, a procedure, a process or a package.

TYPES OF MODULES:

1. **Input:** A module that only produces information that is passed to its superordinate.
2. **Output:** A module that only receives information from its superordinate for output to a device.
3. **Transform:** A module that converts data from one format into another format, possibly generating entirely new information.
4. **Coordinator:** A module that manages the flow of data to and from different sub - modules.
5. **Composite:** Modules that combine one or more of the above styles are composite modules.

OBJECT ORIENTED DESIGN vs FUNCTION ORIENTED DESIGN

OBJECT ORIENTED DESIGN	FUNCTION 0RIENTED DESIGN
• Basic abstraction is not the service available to the users of the system	• Basic abstraction is available to the users of the system
• State information exists in the form of data distributed among several objects of the system	• State information is available in a centralized shared data store
• They constitute a higher level function as a group by grouping functions together	• Group functions together on the basis of the data they operate on
• Modules in the design represent data abstraction	• Consists of module definitions with each module supporting a functional abstraction
• Basic unit of design is Objects	• Basic unit of design is Functions

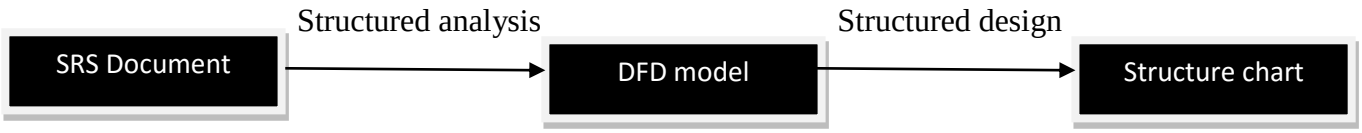
TOP DOWN DESIGN vs BOTTOM UP DESIGN

TOP DOWN DESIGN	BOTTOM UP DESIGN
▪ Using stepwise refinement the whole system is decomposed into subcomponents that exist at lower levels of abstraction.	▪ It starts with primitive components that provide foundational services and using the layers of abstraction builds the functionality, the system needs until entire system has been realized
▪ More useful in situations where specification of the system are clearly known and application is being built from Waterfall model	▪ Useful in situations where a new application is being created from an existing system (Iterative development model

PROBLEM ANALYSIS vs DESIGN PRINCIPLES

PROBLEM ANALYSIS	DESIGN PRINCIPLES
➢ Constructing a model of problem domain	➢ Constructing a model of solution domain
➢ Model depends on the system	➢ System depends on the model
➢ Model is used to understand the problem	➢ Model is used for optimization

STRUCTURED ANALYSIS & STRUCTURED DESIGN METHODOLOGY



- During structured analysis, the SRS document is transformed into a DFD model.
- During structured design, the DFD model is transformed into structure chart.
- Purpose of structured analysis is to capture the detailed structure of the system as perceived by the user, whereas the purpose of structured design is to define the structure of the solution that is suitable for implementation in some programming language.

STRUCTURED ANALYSIS

- The structured analysis activity transforms the SRS document into a graphic model called the DFD model.
- During structural analysis, functional decomposition of system is achieved. That is each function that the system needs to perform is analyzed and hierarchically decomposed into more detailed functions.
- The different functions and data in structured analysis are named using the user's terminology so that the user can ensure that the structured analysis includes all the requirements.
- Structured analysis is based on Top-Down decomposition approach.
- Structured analysis is based on divide and conquer principle.
- The result of structured analysis is a DFD, which is a graphical model of the system that shows the different processing activities or functions that the system performs and the data interchange among those functions.
- DFD represents only the data flow. It doesn't represent any control aspects of the system.

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart. **Transform analysis and Transaction analysis**

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. These are discussed in the subsequent subsections.

STRUCTURE CHART

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks which are used to design structure charts are the following:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

Structure Chart vs. Flow Chart

Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of the software from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

TRANSFORM ANALYSIS

- Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:
  - Input
  - Logical processing
  - Output
- The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an **afferent** branch.
- The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an **efferent** branch. The remaining portion of a DFD is called the **central transform**.
- In the next step of transform analysis, the structure chart is derived by drawing one functional component for the central transform, and the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

- 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 or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central 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 process, identifying customer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

TRANSACTION ANALYSIS

- A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs.
- In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centered system which is characterized by similar processing steps for each data item. Each different way in which input data is handled is a transaction.
- A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions.
- For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module.
- Every transaction carries a tag, which identifies its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module.

DESIGN REVIEW

- After the design is complete, the design is reviewed by a team consist of designers, testers, developers etc.
- The review team checks the design document for the following aspects.
    1. **Traceability:** Whether each bubble of the DFD can be traced to some module in the structure chart and vice versa.
    2. **Correctness:** Whether all the algorithms and data structures of the detailed design are correct.
    3. **Maintainability:** Whether the design can be easily maintained in future.
    4. **Implementation:** Whether the design can be easily and efficiently be implemented.

8.SOFTWARE ARCHITECTURE

- **Definition:** Software architecture is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them.
- A software architecture is an abstraction of a system.
- Architecture defines elements and how they interact.
- Architecture suppresses purely local information about elements; private details are not architectural.
- Externally-visible properties of elements are assumptions that one elements can make about another:

NEED OF SOFTWARE ARCHITECTURE

- Representations of software architecture enable easier communication between all stake holders associated with the system.
- Helps to take earlier design decisions.
- It helps to identify how the system is structured and how its components work together.

USES OF SOFTWARE ARCHITECTURE

- **Understanding and communication:** An architecture description is an important means of communication between various stakes holders. Through this description the stakeholders gain an understanding of the macro properties of the system and how the system intends to fulfil the functional and quality requirements.
- **Reuse:** Architecture descriptions can help software reuse. Reuse will improve productivity, thereby reducing the cost of software. Architecture also facilitates reuse among products that are similar and building product families such that the common parts of these different, but similar products can be reused.
- **Construction and Evolution:** An architecture partitions the system into parts, some architecture provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts. The architecture not only guide the development but also establishes the constraints to be considered during development.
- **Analysis:** It can be used to determine the system behaviour, before the system is actually built. Analysing the architecture helps to determine or predict the properties (quality and performance) of the developing system.

ARCHITECTURAL DESIGN

- Requirements of the software should be transformed into an architecture that describes the software's top-level structure and identifies its components. This is accomplished using architectural design.
- IEEE defines architectural design as ' the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system'.

ARCHITECTURAL STYLES

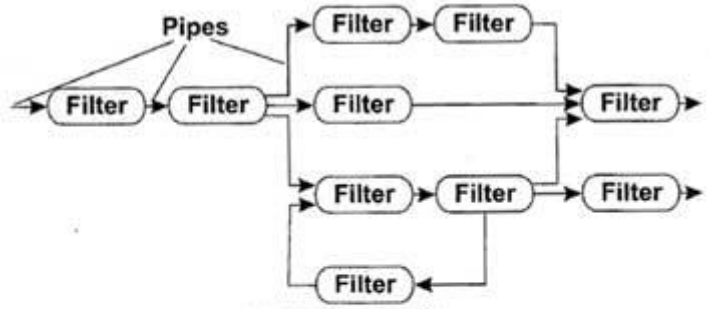
- Architectural styles define a group of interlinked systems that share structural and semantic properties. Every architectural style describes a system category that includes the following.
  - Computational components such as clients, servers, filter and database to execute the desired system function.

- A set of connectors such as procedure calls, pipes, database protocols etc. to provide communication among computational elements.
- Constraints to define integration of components to form a system.

❖ Some of the commonly used architectural styles are data-flow based architecture, object –oriented architecture, layered system architecture, data-centred architecture and call and return architecture.

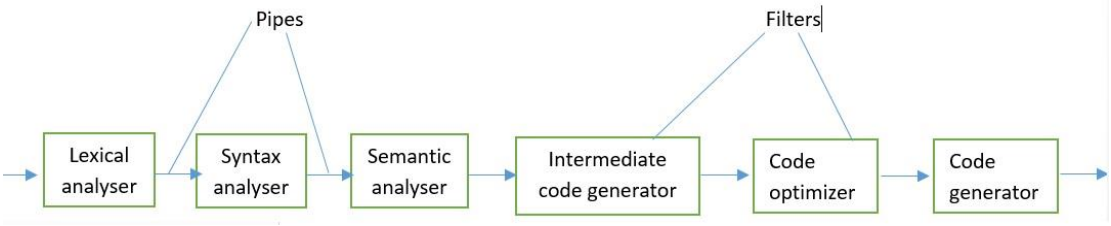
❖ **DATA FLOW ARCHITECTURE / PIPE AND FILTER STYLE**

- Data flow based architecture is mainly used in systems that accepts some inputs and transform it into desired outputs by applying a series of transformations.
- Each component known as **filter**, transforms the data and sends this transformed data to other filters for further processing using the connector, known as **pipe.** Each filter works as an independent entity. A pipe is a unidirectional channel which transports the data received on one end to the other end. Pipe does not change the data in anyway; it merely supplies data to the filter on the receiver end.



Data-flow Architecture

- In most cases, data flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data.
- Example of this style is a Compiler which accepts High Level Language and returns a machine dependent form.



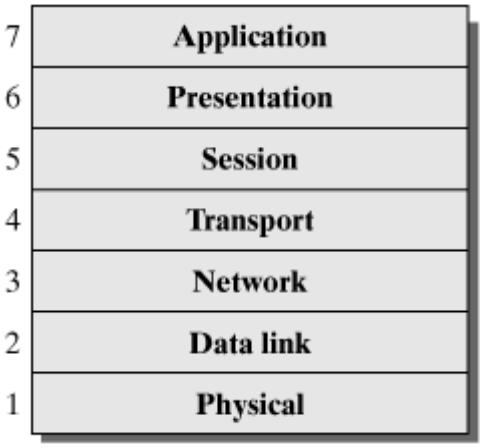
- Advantages of Data- Flow architecture:
  - It supports reusability
  - It is maintainable and modifiable.
  - It supports concurrent execution
- Disadvantages of Data –Flow architecture
  - It often degenerates to batch sequential system.

- It does not provide enough support for applications requires user interaction
- It is difficult to synchronize two different, but related streams.

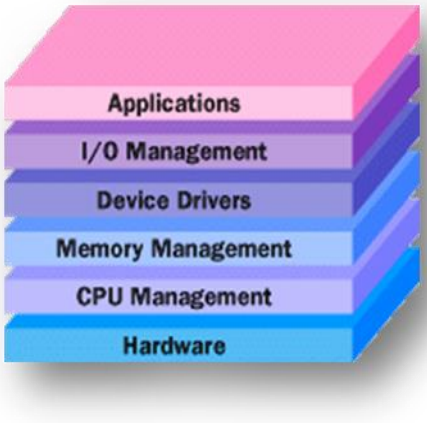
❖ **LAYERED ARCHITECTURE**

- In layered architecture, entire system is divided into various layers such that, each layer performs a well- defined set of operations. These are layers are arranged in hierarchical manner, each one built upon the one below it.
- Each layer provides a set of services to the layer above it and acts as a client to the layer below it.
- The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction.
- Example of layered architectural style is ISO OSI Internet Protocol Suite, Operating system layers etc.

ISO/OSI reference model



Operating system function-layers



❖ **REPOSITORY VIEW / DATA CENTERED VIEW**

- A data centered architecture has two distinct components: a central data store (central repository) and a collection of client software (data accessors).
- The data store (database or file) represents the current state of the data and the client software performs several operations like add, delete, update etc., on the data stored in the central repository.
- It has two views:

  Repository view: The client who updates the central repository can determine the access permission for various other clients, to view the same updated data.

  Black board view: Here the data store acts like a black board system in which the data store is transformed into a black board that notifies the client software when the data changes / updates.

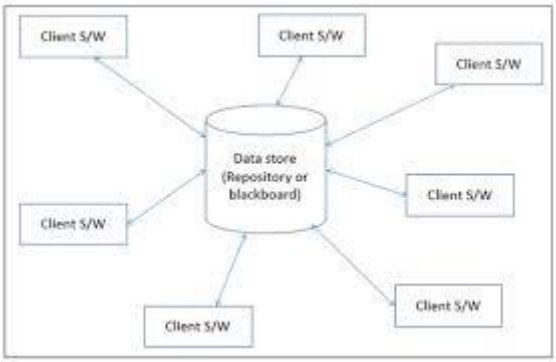


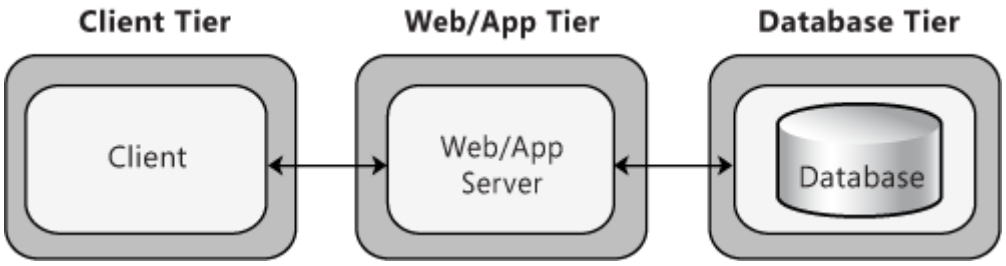
Figure: Repository view

- **ADVANTAGES:**
  - Clients operate independently of one another.
  - Data repository is independent of clients.
  - It facilitates scalability.
  - Supports modifiability.

❖ CLIENT – SERVER STYLE

- It is a variant of main program and subroutines concept, but the clients and servers.
- A server provides different services. A client uses services as (remote) subroutines from one or more servers.
- A general form of this architecture is n – tier architecture. In this style, a client sends request to a server. But in order to service the request, the server sends some request to another server. That is the server acts as a client for the next tier. This hierarchy can continue for some levels, providing a n-tier system.

- A common example is 3 –tier architecture.

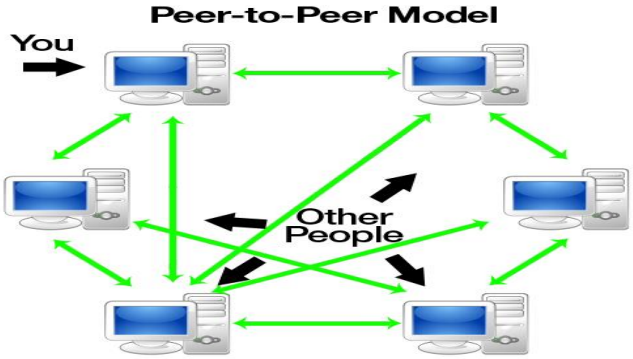


❖ CALL AND RETURN ARCHITECTURE

- This architecture enables to achieve a program structure that is relatively easy to modify and scale. The categories are Main program/subprogram architecture and Remote Procedure Call (RPC) architectures.

❖ **PEER – PEER ARCHITECTURE / OBJECT ORIENTED VIEW**

- Peer to Peer (P2P) are similar to client – server but all processes can be act as clients and servers.
- It is more flexible but complicated design.
- There will be chance to occur dead lock or starvation.



COMPONENT LEVEL DESIGN

- Component-level design occurs after the first iteration of the architectural design.
- It strives to create a design model from the analysis and architectural models.
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code.
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors.
- A software component is a modular building block for computer software. It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with other components or entities outside the boundaries of the system
- Three different views of a component
    1. **An object-oriented view:** A component is viewed as a set of one or more collaborating classes *(Object Oriented Design)*
    2. **A conventional view**: A component is viewed as a functional element (i.e., a module) of a program that incorporates functional logic, internal data structures and interfaces. *(Function Oriented design)*
    3. **A process-related view:** Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch.

FUNCTION ORIENTED DESIGN

- In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
- Function oriented design inherits some properties of structured design where divide and conquer methodology is used. Uses top – down approach
- This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

- Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process in Function Oriented Design:

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions change data and state of the entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

OBJECT ORIENTED DESIGN

- ❖ Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.
- ❖ The various concepts of Object Oriented Design is as follows:

1. **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
2. **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object. In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
3. **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
4. **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
5. **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Design Process in Object Oriented Design

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them is defined.
- Application framework is defined.

USER INTERFACE DESIGN / INTERFACE DESIGN

- User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.
- User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.
- UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.
- The software becomes more popular if its user interface is:

1. Attractive
2. Simple to use
3. Responsive in short time
4. Clear to understand
5. Consistent on all interfacing screens

CHARACTERISTICS OF A GOOD USER INTERFACE

- **Speed of learning:** A good user interface should not require its users to memorize commands. It should be easy for the user to learn the actions to be performed.
- **Speed of use / reuse:** Speed of use of a UI is determined by the time and user effort necessary to initiate and execute different commands. A good UI should provide the facility to reuse the recently used commands.
- **Speed of recall:** Once users Learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This can be achieved by using suitable metaphors, command issue procedures etc.
- **Error prevention:** A good UI should minimize the scope of committing errors while initiating different commands.
- **Aesthetic and attractive:** A good UI should be attractive to use. GUI based interfaces are more aesthetic and attractive than Text based user interfaces.
- **Consistency:** The notations / icons / commands should be consistent. It should be relevant or relative to the operation they are performing. It does not create any ambiguity in user's mind.
- **Feedback:** A good UI should provide a feed-back message to the user for every user action.
- **Support for multiple user categories:** A good UI should provide the facility to support the various levels of users. An experienced user and a novice user can easily use the software without much burden.
- **Error recovery (Undo facility):** A good UI should provide the feature to correct the mistakes committed by the users. Eg: Undo option
- **User guidance and online help:** User may not be aware of every aspects or states of the system. So a good UI should provide the facility to show user guidance messages as well as online support.

TYPES OF USER INTERFACES

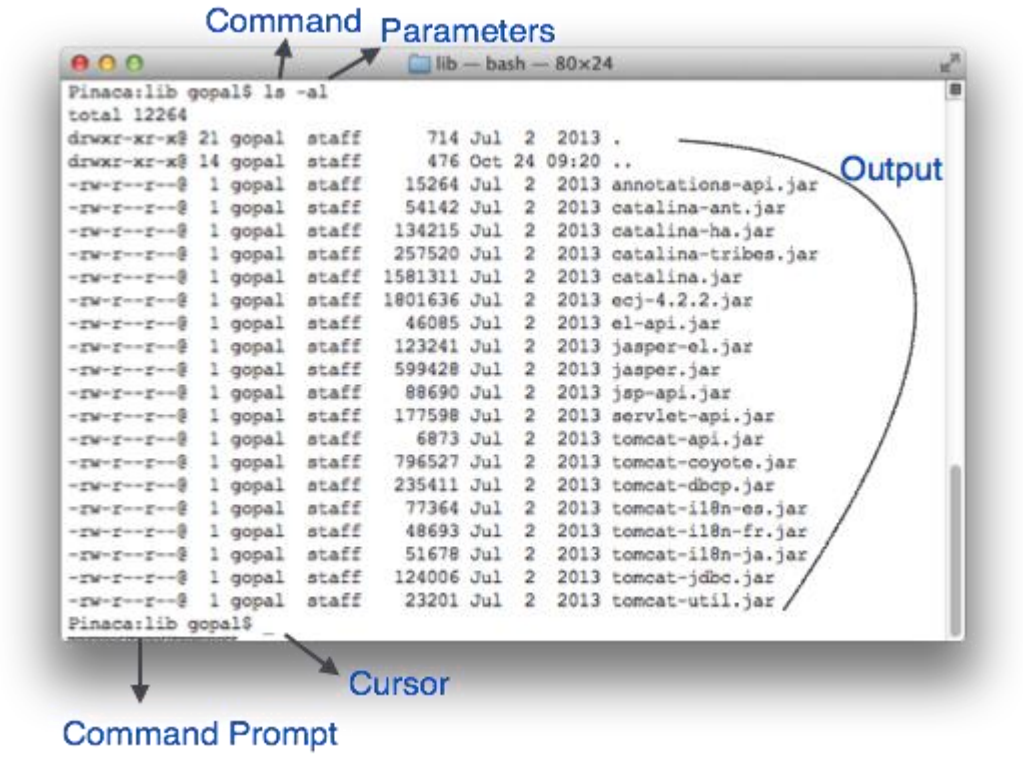
UI is broadly divided into two categories:

- Command Line Interface / Text based interface
- Graphical User Interface

TEXT BASED INTERFACE / Command Line Interface (CLI)

- CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.
- CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.
- A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.
- CLI uses less amount of computer resource as compared to GUI.

CLI Elements



A text-based command line interface can have the following elements:

- **Command Prompt** - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
- **Cursor** - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
- **Command** - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

GRAPHICAL USER INTERFACE

- Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.
- Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

GUI Elements

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:

- **Window** - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.
- **Tabs** - If an application allows executing multiple instances of itself, they appear on the screen as separate windows. **Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.
- **Menu** - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.
- **Icon** - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.

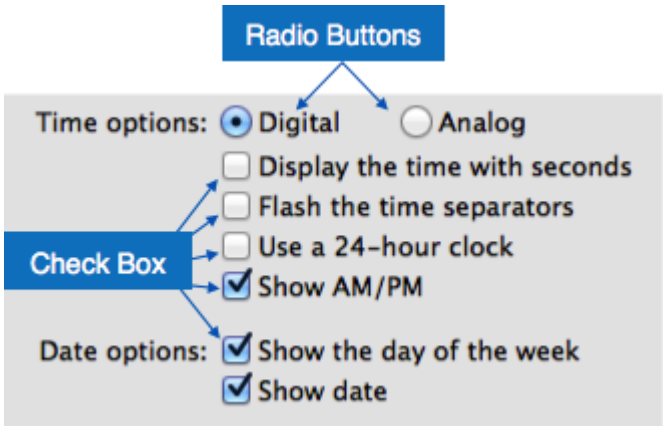
- **Cursor** - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

Application specific GUI components

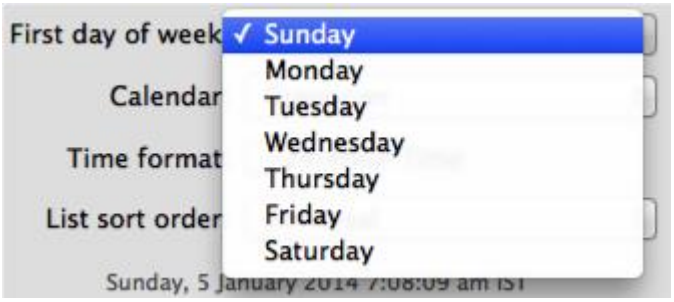
A GUI of an application contains one or more of the listed GUI elements:

- **Application Window** - Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.
- **Dialogue Box** - It is a child window that contains message for the user and request for some action to be taken. For Example: Application generate a dialogue to get confirmation from user to delete a file.

- **Text-Box** - Provides an area for user to type and enter text-based data.
- **Buttons** - They imitate real life buttons and are used to submit inputs to the software.



- **Radio-button** - Displays available options for selection. Only one can be selected among all offered.
- **Check-box** - Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.
- **List-box** - Provides list of available items for selection. More than one item can be selected.



Other impressive GUI components are:

- Sliders
- Combo-box
- Data-grid
- Drop-down list

GUI DESIGN METHODOLOGY

- There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.
- The entire design methodology consist of the following three phases:

    1. Analysis & Modelling
    2. Interface design
    3. Interface implementation and validation

- The above said phases are sub divided into the following set of activities as listed below:
- **GUI Requirement Gathering** - The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
- **User Analysis** - The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
- **Task Analysis** - Designers have to analyse what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
- **GUI Design & implementation** - Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.
- **Testing / Evaluation** - GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

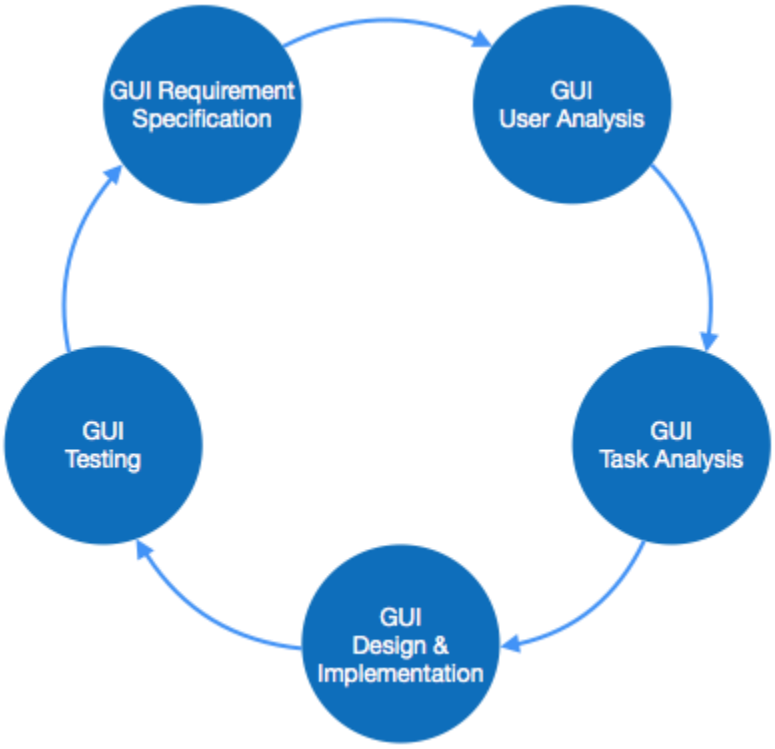


Figure: GUI Design Methodology

UI - EVALUATION TECHNIQUES

- **Use it yourself:** This is the first technique of evaluation, where the designer himself uses the interface for a period of time to evaluate its good and bad features.
- **Colleague evaluation:** Designers are aware of the functionalities of the software, there is a chance to miss out the issues of ease of learning and efficiency. Evaluating the UI by a colleague in the organization may help in solving these issues.
- **User testing:** Considered as the most practical approach of evaluation. Here the user test the prototype and provides feedback to the designer. Designer modifies the prototype based on customer / user feedback.
- **Heuristic evaluation:** The UI is evaluated by a team of experienced designers. Experienced designers can identify more mistakes than a less experienced designer.

MODE BASED vs MODELESS INTERFACE

MODE BASED INTERFACE	MODELESS BASED INTERFACE
Same set of commands can be invoked at any time during the running of the software	Different set of commands can be invoked, depending on the mode in which the system is.
Has only a single mode and all commands are available all the time during the operation of the software	Mode at any instant is determined by the sequence of commands already issued by the user.

GUI vs TEXT BASED INTERFACE

CRITERIA	TEXT BASED	GUI BASED
EASE	Because of the memorization and familiarity needed to operate a command line interface, new users have a difficult time navigating and operating a command line interface.	Although new users may have a difficult time learning to use the mouse and all GUI features, most users pick up this interface much easier when compared to a command line interface.
CONTROL	Users have much more control of their file system and operating system in a command line interface. For example, users can copy a specific file from one location to another with a one-line command.	Although a GUI offers plenty of control of a file system and operating system, the more advanced tasks may still need a command line.
MULTITASKING	Although many command line environments are capable of multitasking, they do not offer the same ease and ability to view multiple things at once on one screen.	GUI users have windows that enable a user to view, control, and manipulate multiple things at once and is much faster to navigate when compared with a command line.
SPEED	Command line users only need to use their keyboards to navigate a command line interface and often only need to execute a few lines to perform a task	A GUI may be easier to use because of the mouse. However, using a mouse and keyboard to navigate and control your operating system for many things is going to be much slower than someone who is working in a command line.
RESOURCES	A computer that is only using the command line takes a lot less of the computer's system resources than a GUI.	A GUI requires more system resources because of each of the elements that need to be loaded such as icons, fonts, etc. In addition, video drivers, mouse drivers, and other drivers that need to be loaded take additional system resources.
SCRIPTING	A command line interface enables a user to script a sequence of commands to perform a task or execute a program.	Although A GUI enables a user to create shortcuts, tasks, or other similar actions, it doesn't even come close in comparison to what is available through a command line.
REMOTE ACCESS	When accessing another computer or networking device over a network, a user can only manipulate the device or its files using a command line interface.	Although remote graphical access is possible. Not all computers and especially not all network equipment has this ability.
DIVERSE	After you've learned how to navigate and use a command line, it's not going to change as much as a new GUI. Although new commands may be introduced, the original commands always remain the same.	Each GUI has a different design and structure of how to perform different tasks. Even different versions of the same GUI, such as Windows, can have hundreds of different changes between each version.
STRAIN	More strain for the user.	Shortcuts reduces the strain of the user.

9. SOFTWARE MAINTENANCE

- Software maintenance denotes any changes made to a software product after it has been delivered to the customer.

- *Software maintenance* in *software* engineering is the modification of a *software* product after delivery to correct faults, to improve performance or other attributes. A common perception of *maintenance* is that it merely involves fixing defects.

Types of software maintenance

- **Corrective:** Correcting the errors that are not identified during the product development phase. Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.
- **Adaptive:** Maintenance done for porting the software to work in a new environment (For example, new Operating system or new hardware platform etc.) A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** Those changes made for improving the implementation of the system and enhancing the system functionalities according to the client's requirements. A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.
- **Preventive:** Those changes made to the software to prevent the chance of occurring errors.

Characteristics of software evolution / Laws of Software maintenance

Lehman and Belady have studied the characteristics of evolution of several software products and formulated the following laws.

1. **Lehman's First law:** A software product must change continually or become progressively less useful.
2. **Lehman's Second law:** The structure of a software / program tends to degrade as more and more maintenance is carried out on it.
3. **Lehman's Third law:** Over a program's life time, its rate of development is approximately constant.

Problems associated with software maintenance

- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation / Difficult to carry out maintenance on legacy products.
- The work involved in maintenance process is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

SOFTWARE REVERSE ENGINEERING

- Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.
- The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.
- The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities.
- A process model for reverse engineering has been shown below.

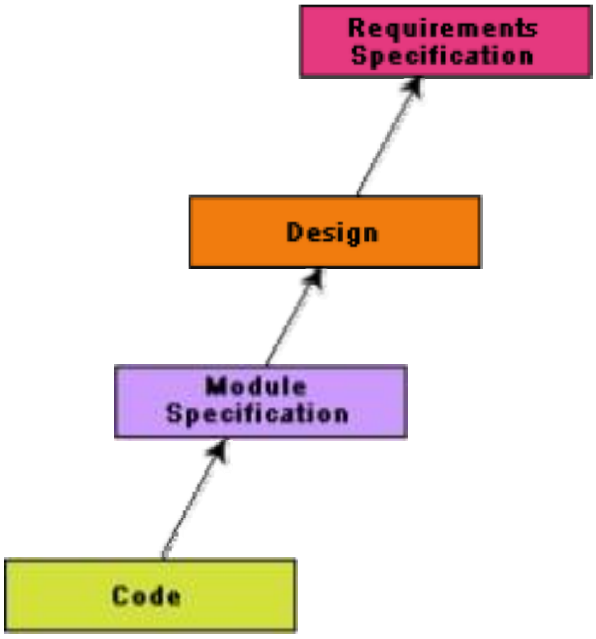


Fig: A process model for reverse engineering

- Programs are reformulated to identify the requirements. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend.
- Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible.
- Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.
- After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in figure below.

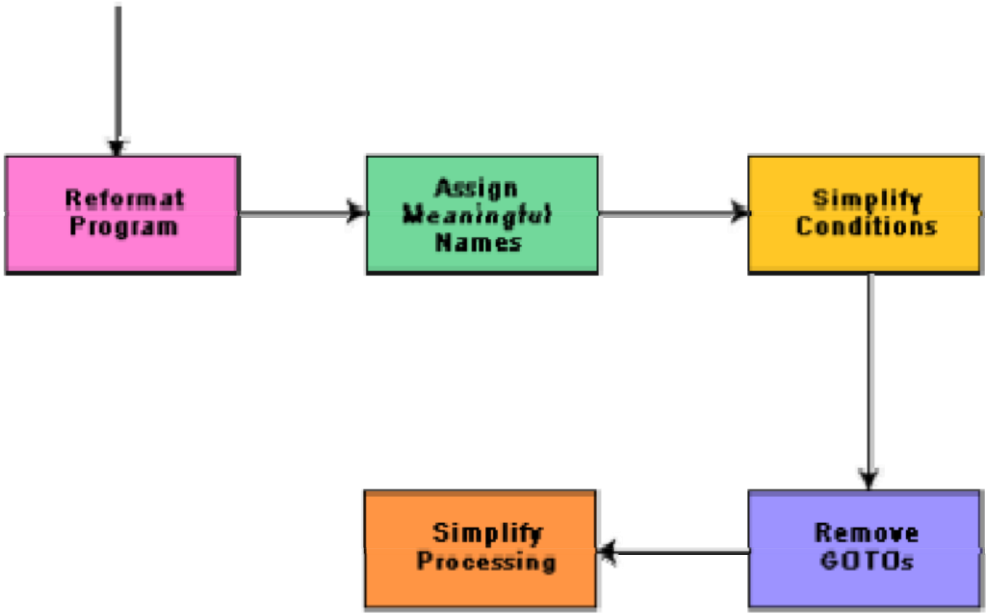


Figure: Cosmetic changes carried out before reverse engineering

- In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code.
- The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

Legacy software products

- It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product.
- Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

Factors on which software maintenance activities depend

The activities involved in a software maintenance project are not unique and depend on several factors such as:

- the extent of modification to the product required
- the resources available to the maintenance team
- the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.)
- the expected project risks, etc.

SOFTWARE MAINTENANCE PROCESS MODELS

- When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a

reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

- Two broad categories of process models for software maintenance can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented below.

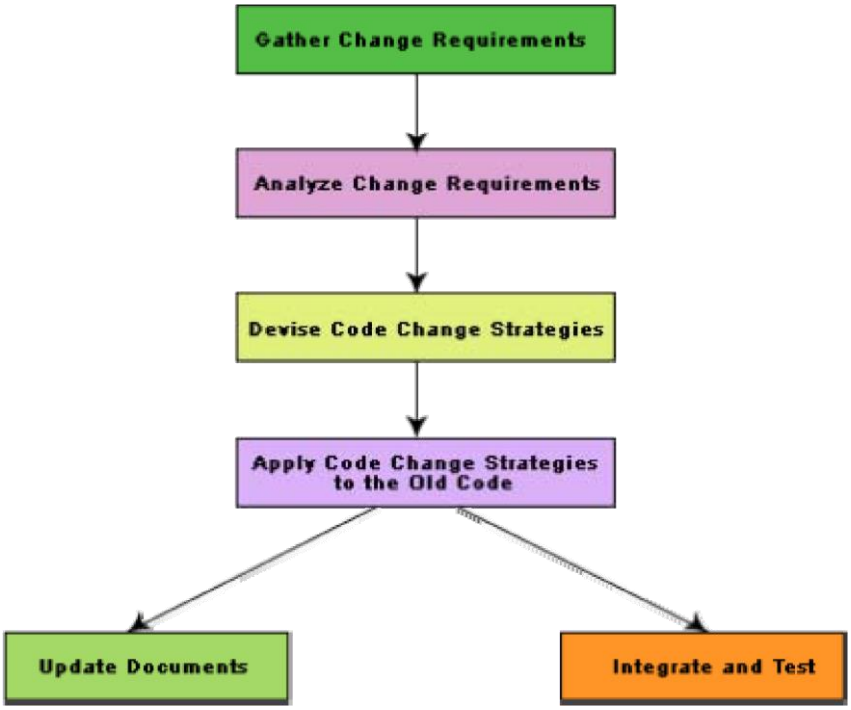


Figure: Maintenance process model 1

- In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle team, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.
- The second process model for software maintenance is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software reengineering. This process model is depicted in following figure.

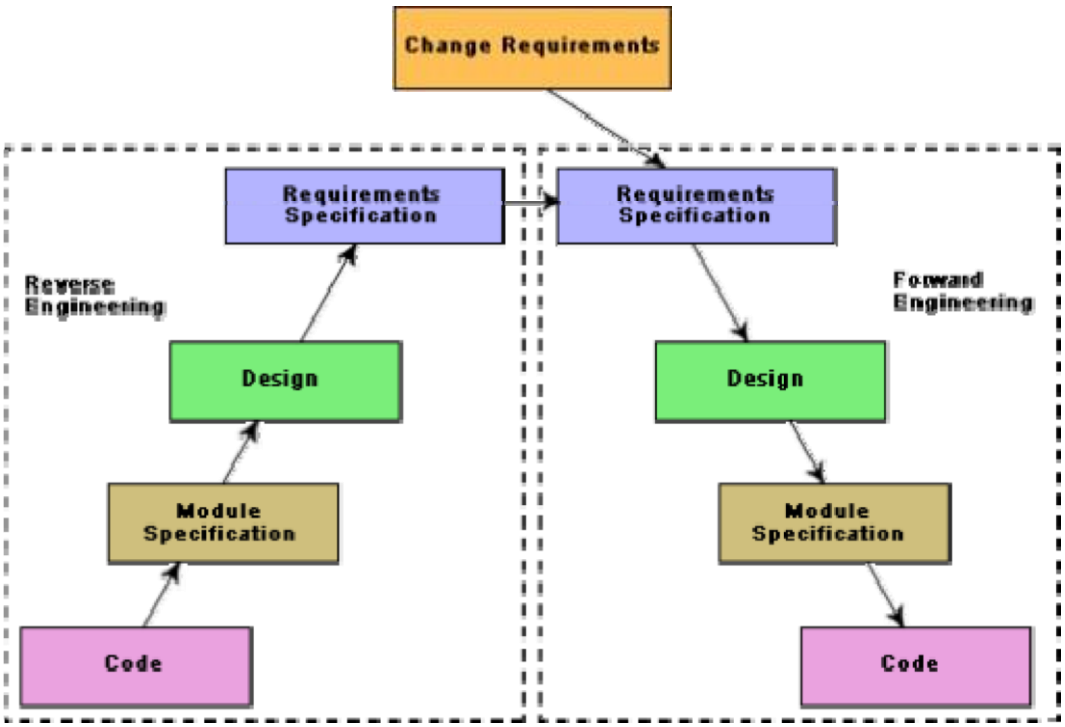


Figure: Maintenance process model 2

- The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications. The module specifications are then analyzed to produce the design.
- The design is analyzed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification.
- At the design, module specification, and coding a substantial reuse is made from the reverse engineered products.
- An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15% (as shown in fig. 14.5). Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2:

  - Reengineering might be preferable for products which exhibit a high failure rate.
  - Reengineering might also be preferable for legacy products having poor design and code structure.

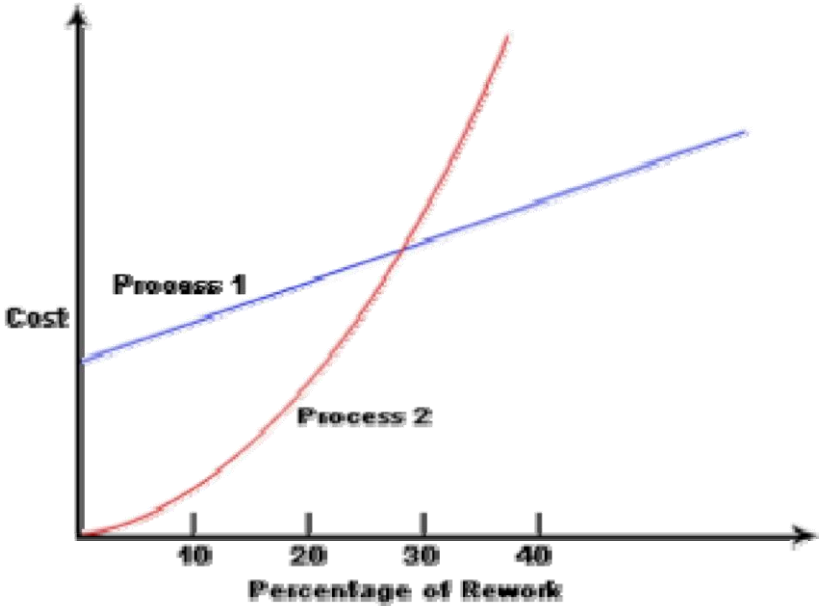


Fig. 14.5: Empirical estimation of maintenance cost versus percentage rework

Software reengineering

Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering.

Estimation of approximate maintenance cost

- It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.
- Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

where, K^{LOC}_{added} is the total kilo lines of source code added during maintenance.

$K^{LOC}_{deleted}$ is the total KLOC deleted during maintenance.

Thus, the code that is changed, should be counted in both the code added and the code deleted. The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

Maintenance cost = *ACT* × development cost.

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

SOFTWARE REUSE

Software reuse, also called code reuse, is the use of existing software, or software knowledge, to build new software, following the reusability principles.

REUSABLE COMPONENT TYPES:

- **Application system reuse**

The whole of an application system may be reused on a different machine. Usually referred to as program portability

- **Modules or object reuse**

The reusable component is a collection of functions or procedures + its data structures…

- **Function reuse**

The reusable component is a single function

REUSABLE SOFTWARE COMPONENTS / REUSABLE ARTIFACTS

The various reusable components are:

- Requirements specification
- Design
- Code
- Test cases
- Knowledge

ADVANTAGES OF SOFTWARE REUSE

- System reliability is increased
- Overall risk is reduced
- Effective use can be made of specialists
- Organizational standards can be embodied in reusable components
- Software development time can be reduced

BASIC ISSUES IN ANY REUSE PROGRAM

- **Component creation.** For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. Domain analysis is a promising technique which can be used to create reusable components.
- **Component indexing and storing.** Indexing requires classification of the reusable components so that they can be easily searched when looking for a component for reuse. The components need to be stored in a Relational Database Management System (RDBMS) or an Object-Oriented Database System (ODBMS) for efficient access when the number of components becomes large.
- **Component searching.** The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.
- **Component understanding.** The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.

- **Component adaptation.** Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.
- **Repository maintenance.** A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository. The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

REUSE APPROACH

- **Domain analysis**

The aim of domain analysis is to identify the reusable components for a problem domain.

- A reuse domain is a technically related set of application areas. A body of information is considered to be a problem domain for reuse, if a deep and comprehensive relationship exists among the information items as categorized by patterns of similarity among the development components of the software product.
- Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.
- During domain analysis, a specific community of software developers gets together to discuss community-wide-solutions. Analysis of the application domain is required to identify the reusable components. The actual construction of reusable components for a domain is called domain engineering.

- **Component classification & Searching**
  - The domain repository may contain thousands of reuse items. A popular search technique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results do a browsing using the links provided to look up related items.
  - The approximate automated search locates products that appear to fulfil some of the specified requirements. The items located through the approximate search serve as a starting point for browsing the repository. These serve as the starting point for browsing the repository.
  - The developer may follow links to other products until a sufficiently good match is found. Browsing is done using the keyword-to-keyword, keyword-to-product, and product-to-product links. These links help to locate additional products and compare their detailed attributes.
  - Finding a satisfactorily item from the repository may require several locations of approximate search followed by browsing. With each iteration, the developer would get a better understanding of the available products and their differences. Search items includes components, designs, models, requirements, and even knowledge.
- **Repository maintenance**
  - Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search.
  - The software industry is always trying to implement something that has not been quite done before. As patterns requirements emerge, new reusable components are identified, which may ultimately become more or less the standards. However, as technology advances, some components which are still reusable, do not fully address the current requirements.
- **Reuse without modifications**

- Once standard solutions emerge, no modifications to the program parts may be necessary. One can directly use the parts to develop his application. Reuse without modification is more useful than the classical library modules.

10. SOFTWARE TESTING

Software testing is a process of executing a program or application with the intent of finding the software bugs.

- ✓ It can also be stated as the **process of validating and verifying** that a software program or application or product:
  - Meets the business and technical requirements that guided it's design and development
  - Works as expected
  - Can be implemented with the same characteristic.

- ✓ **Verification:** The set of activities that ensure that software correctly implements a specific function or algorithm. (Are the algorithms coded correctly?) (Are we building the product right?)
- ✓ **Validation:** The set of activities that ensure that the software that has been built is traceable to customer requirements. (Does it meet user requirements?) (Are we building the right product?)

NEED OF SOFTWARE TESTING

1. Software testing is really required to point out the defects and errors that were made during the development phases.
2. It's essential since it makes sure of the Customer's reliability and their satisfaction in the application.
3. It is very important to ensure the Quality of the product. Quality product delivered to the customers helps in gaining their confidence.
4. Testing is necessary in order to provide the facilities to the customers like the delivery of high quality product or software application which requires lower maintenance cost and hence results into more accurate, consistent and reliable results.
5. Testing is required for an effective performance of software application or product.
6. It's important to ensure that the application should not result into any failures because it can be very expensive in the future or in the later stages of the development.
7. It's required to stay in the business.

TERMINOLOGIES IN TESTING

- **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.
- **Test suite:** This is the set of all test cases with which a given software product is to be tested.
- **Error:** Error is deviation from actual and expected value. It represents mistake made by people.
- **Fault:** Fault is incorrect step, process or data definition in a computer program which causes the program to behave in an unintended or unanticipated manner. It is the result of the error.

- **Bug:** Bug is a fault in the program which causes the program to behave in an unintended or unanticipated manner. It is an evidence of fault in the program.

- **Failure:** Failure is the inability of a system or a component to perform its required functions within specified performance requirements. Failure occurs when fault executes.

- **Defect:** A defect is an error in coding or logic that causes a program to malfunction or to produce incorrect/unexpected results. A defect is said to be detected when a failure is observed.

OBJECTIVES OF SOFTWARE TESTING

- Finding defects which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of quality.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

COMMON ERRORS TO UNCOVER DURING TESTING

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)
- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using == with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

TESTING ACTIVITIES

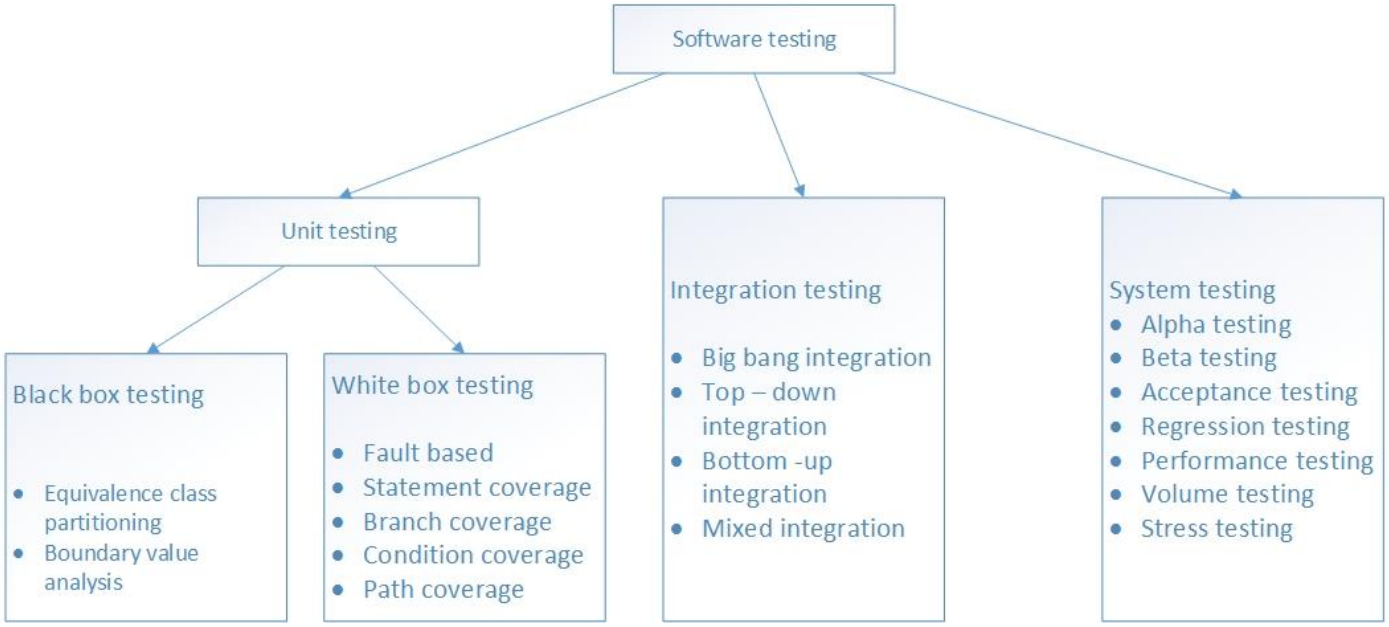
1. **Test Suite design**

2. **Running test cases and checking the result to detect failures*:*** *Each test case is run and the results are compared with expected results.*

3. **Debugging:** *To identify the statements that are in error.*

4. **Error correction:** *Code is appropriately changed to correct the error.*

TYPES OF TESTING



UNIT TESTING

- A unit is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.
- **Unit tests are basically written and executed by software developers** to make sure that code meets its design and requirements and behaves as expected.
- The goal of unit testing is to segregate each part of the program and test that the individual parts are working correctly.
- This means that for any function or procedure when a set of inputs are given then it should return the proper values. It should handle the failures gracefully during the course of execution when any invalid input is given.
- Unit testing should be done before Integration testing. Unit testing should be done by the developers.
- A unit test provides a written contract that the piece of code must assure. Hence it has several benefits.

Advantages of Unit testing:

1. Issues are found at early stage. Since unit testing are carried out by developers where they test their individual code before the integration. Hence the issues can be found very early and can be resolved then and there without impacting the other piece of codes.

2. Unit testing helps in maintaining and changing the code. This is possible by making the codes less interdependent so that unit testing can be executed. Hence chances of impact of changes to any other code gets reduced.

3. Since the bugs are found early in unit testing hence it also helps in reducing the cost of bug fixes. Just imagine the cost of bug found during the later stages of development like during system testing or during acceptance testing.

4. Unit testing helps in simplifying the debugging process. If suppose a test fails then only latest changes made in code needs to be debugged.

BLACK BOX TESTING / FUNCTIONAL TESTING / BEHAVIOURAL TESTING

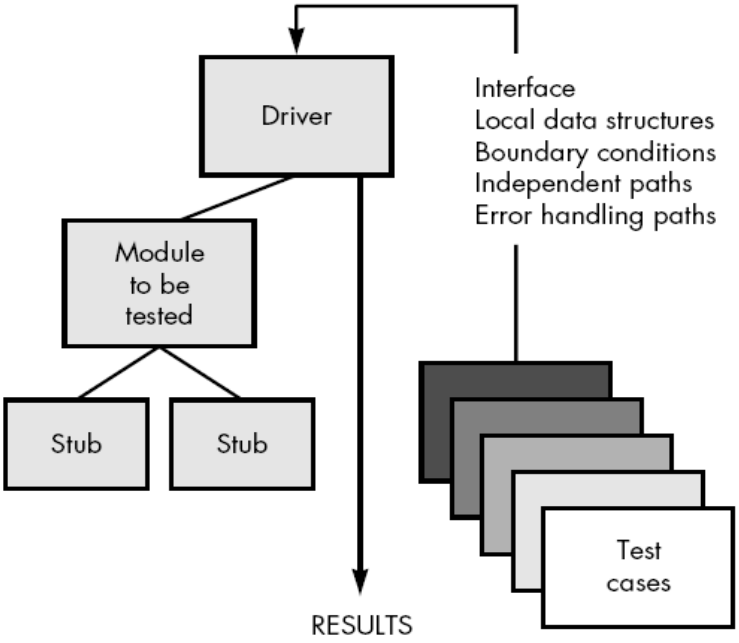
- Ensures the functionality of the system or program.
- Specification-based testing technique is also known as **'black-box'** or input/output driven testing techniques because they view the software as a black-box with inputs and outputs.
- The testers have no knowledge of how the system or component is structured inside the box. In black-box testing the tester is concentrating on what the software does, not how it does it.
- The definition mentions both functional and non-functional testing. Functional testing is concerned with what the system does its features or functions. Non-functional testing is concerned with examining how well the system does. Non-functional testing like performance, usability, portability, maintainability, etc.
- Specification-based techniques are appropriate at all levels of testing (component testing through to acceptance testing) where a specification exists. For example, when performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests.

- There are four specification-based or black-box technique:

    - Equivalence partitioning
    - Boundary value analysis
    - Decision tables
    - State transition testing

DRIVER AND STUB MODULES

- **Driver**
    - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- **Stubs**
    - Serve to replace modules that are subordinate to (called by) the component to be tested
    - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing



- ✓ Stubs and drivers are used to replace the missing software and simulate the interface between the software components in a simple manner.
- ✓ **For example:** Suppose you have a function (Function A) that calculates the total marks obtained by a student in a particular academic year. Suppose this function derives its values from another function (Function b) which calculates the marks obtained in a particular subject. You have finished working on Function A and wants to test it. But the problem you face here is that you can't seem to run the Function A without input from Function B; Function B is still under development. In this case, you create a dummy function to act in place of Function B to test your function. This dummy function gets called by another function. Such a dummy is called a Stub. To understand what a driver is, suppose you have finished Function B and is waiting for Function A to be developed. In this case you create a dummy to call the Function B. This dummy is called the driver.

EQUIVALENCE CLASS PARTITIONING

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once
- The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes – the two terms mean exactly the same thing.
- In equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a

partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

GUIDELINES FOR DERIVING EQUIVALENCE CLASSES

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false condition}

BOUNDARY VALUE ANALYSIS

- Boundary value analysis (BVA) is based on testing at the boundaries between partitions.
- Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).
- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
- It selects test cases at the edges of a class
- It derives test cases from both the input domain and output domain
- GUIDELINES FOR DERIVING EQUIVALENCE CLASSES

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

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

Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above

If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

- For example: An input field (Date of BIRTH) in which **Month** as an input data. Then the valid input should be any integer from 1 to 12, since there are only 12 months in a calendar year. So if the user enters the data as '25 ', then the system shouldn't accept it. For that we should write the program condition as (1<=month<=12)
- So in the above case, in **Equivalence class partitioning**, there are three classes. One valid class and two invalid class.
  Valid class: Any number in between 1 and 12 (1<=month<=12)
  Invalid class 1: Numbers less than 1 (Month<1)
  Invalid class 2: Numbers greater than 12 (Month>12)

So we should select a test case from each of the three equivalence classes. So the test suite may contain a number between 1 and 12, a number less than 1 and a number greater than 12.
Test suite = {6, -12, 30}

- So in the above case, in **Boundary value analysis**, the boundary values are 1 and 12. So choose a number just below the minimal condition value and choose a value just above the maximum condition value.
  So we should select the number just below 1 and just above 12 along with 1 and 12.
  Test Suite = {0,1,12,13}

WHITE BOX TESTING / STRUCTURAL TESTING / GLASS BOX TESTING

- Structure-based testing technique is also known as **'white-box'** or 'glass-box' testing technique because here the testers require knowledge of how the software is implemented, how it works.
- Focuses on the internal structure of the software or program.
- In white-box testing the tester is concentrating on how the software does it. For example, a structural technique may be concerned with exercising loops in the software.
- Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.
- Structure-based techniques can also be used at all levels of testing. Developers use structure-based techniques in component testing and component integration testing, especially where there is good tool support for code coverage.
- Structure-based techniques are also used in system and acceptance testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.
- Uses two testing strategies: Fault based testing and Coverage based testing.

  - **Fault based testing:** It targets to detect certain types of faults. Mutation testing is a kind of fault based testing.
  - **Coverage based testing:** It attempts to execute (or cover) certain elements of a program. The various coverage based strategies are statement, branch, condition, path coverage-based testing.

STATEMENT COVERAGE

- Statement coverage based strategy aims to design test cases to ensure that , every statement is executed at least once.
- The statement coverage is also known as line coverage or segment coverage.
- The principal idea in statement coverage is that unless a statement is executed, there is no way to determine whether an error exists in that statement.
- The statement coverage **covers only the true conditions.**
- Through statement coverage we can identify the statements executed and where the code is not executed because of blockage.
- In this process each and every line of code needs to be checked and executed

Advantage of statement coverage:

- It verifies what the written code is expected to do and not to do
- It measures the quality of code written
- It checks the flow of different paths in the program and it also ensure that whether those path are tested or not.

Disadvantage of statement coverage:

- It cannot test the false conditions.
- It does not report that whether the loop reaches its termination condition.
- It does not understand the logical operators.

The statement coverage can be calculated as shown below:

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

BRANCH COVERAGE

- Decision coverage also known as branch coverage or all-edges coverage or edge testing.
- It **covers both the true and false conditions** unlikely the statement coverage.
- A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested. This sounds great because it takes a more in-depth view of the source code than simple statement coverage
- A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement. With an IF statement, the exit can either be TRUE or FALSE, depending on the value of the logical condition that comes after IF.

Advantages of decision coverage:

- To validate that all the branches in the code are reached
- To ensure that no branches lead to any abnormality of the program's operation
- It eliminate problems that occur with statement coverage testing

Disadvantages of decision coverage:

- This metric ignores branches within boolean expressions which occur due to short-circuit operators.

The decision coverage can be calculated as given below:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

CONDITION COVERAGE

- This is closely related to decision coverage but has better sensitivity to the control flow.
- Test cases are designed to make each component of composite conditional expression to assume both true and false values.
- For a composite conditional expression with '**n**' components, there will be $\mathbf{2^n}$ test cases required to ensure condition coverage.

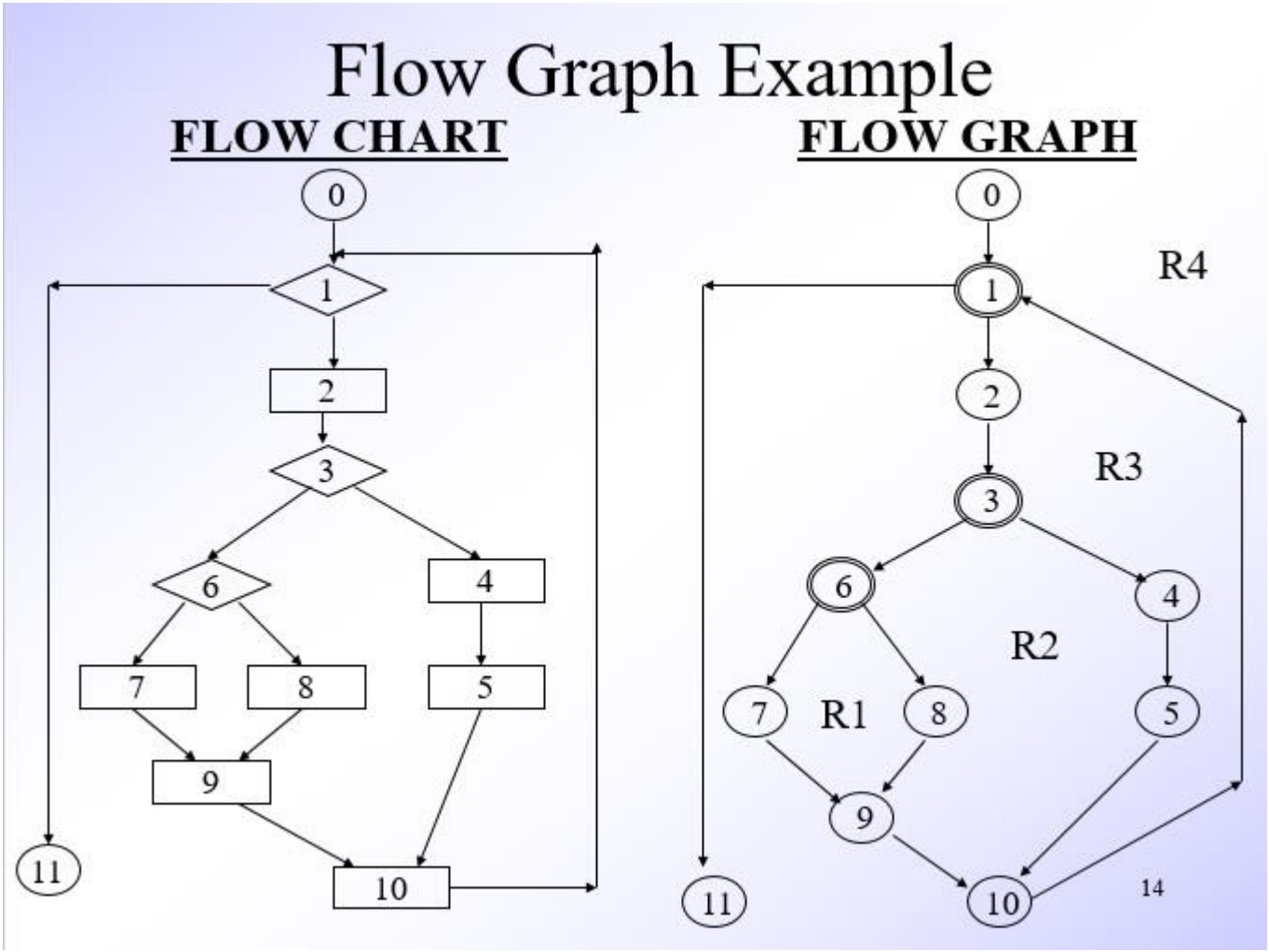
- However, full condition coverage does not guarantee full decision coverage.
- Condition coverage reports the true or false outcome of each condition.
- Condition coverage measures the conditions independently of each other.

PATH COVERAGE

- White-box testing technique proposed by Tom McCabe.
- It ensures that all linearly independent paths (or basis paths) in the program are executed at least once.
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

CFG NOTATION

- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too



- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along at least one edge that has not been traversed before by a previous path
- Basis set for flow graph on the above figure.
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

❖ **MCCABE'S CYCLOMATIC COMPLEXITY**

- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - The number of regions / No. of closed regions + 1

- V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
- V(G) = P + 1, where P is the number of predicate nodes in the flow graph G

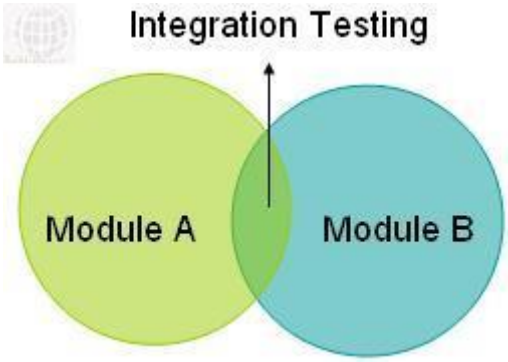
- Results in the following equations for the example flow graph shown in previous page.
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

❖ **STEPS FOR DERIVING BASIS SET & TEST CASES**

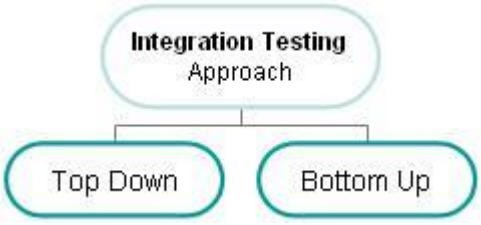
1. Using the design or code as a foundation, draw a corresponding flow graph
2. Determine the cyclomatic complexity of the resultant flow graph
3. Determine a basis set of linearly independent paths
4. Prepare test cases that will force execution of each path in the basis set

INTEGRATION TESTING

- Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems.
- Also after integrating two different components together we do the integration testing. As displayed in the image below when two different modules 'Module A' and 'Module B' are integrated then the integration testing is done.



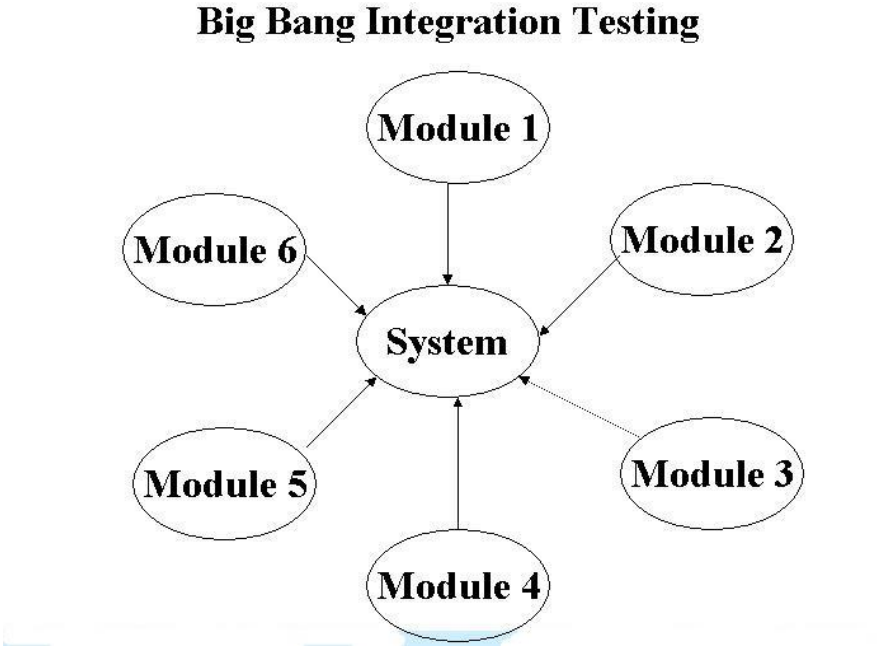
- Integration testing is done by a specific integration tester or test team.
- Integration testing follows two approach known as 'Top Down' approach and 'Bottom Up' approach as shown in the image below:



INTEGRATION TESTING TECHNIQUES

1. Big Bang integration testing:

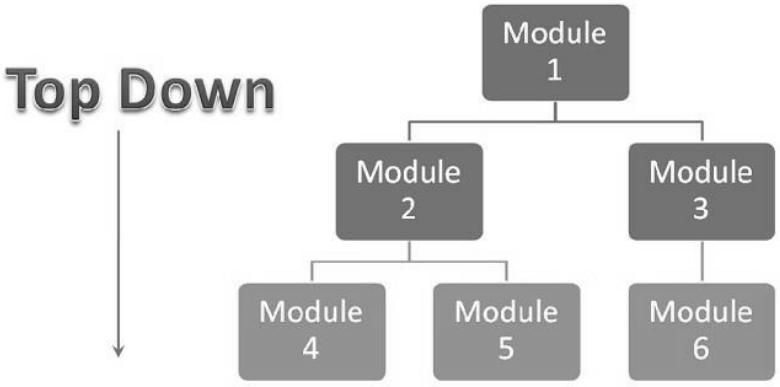
In Big Bang integration testing all components or modules are integrated simultaneously, after which everything is tested as a whole. As per the below image all the modules from 'Module 1′ to 'Module 6′ are integrated simultaneously then the testing is carried out.



Advantage: Big Bang testing has the advantage that everything is finished before integration testing starts.

Disadvantage: The major disadvantage is that in general it is time consuming and difficult to trace the cause of failures because of this late integration.

2. **Top-down integration testing:** Testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs. Below is the diagram of 'Top down Approach':



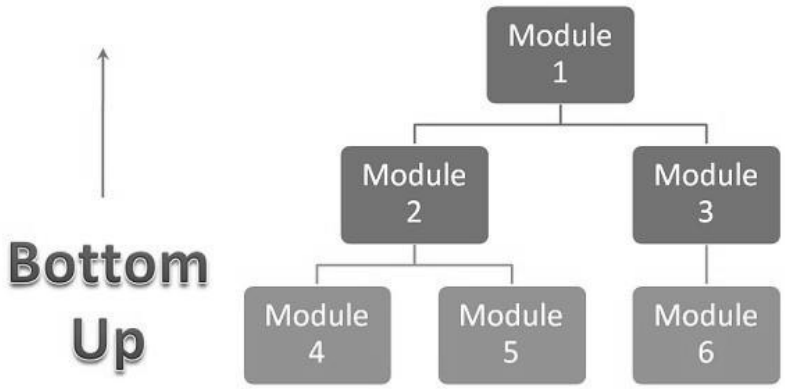
Advantages of Top-Down approach:

- The tested product is very consistent because the integration testing is basically performed in an environment that almost similar to that of reality
- Stubs can be written with lesser time because when compared to the drivers then Stubs are simpler to author.

Disadvantages of Top-Down approach:

- Basic functionality is tested at the end of cycle

3. **Bottom-up integration testing:** Testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers. Below is the image of 'Bottom up approach':



Advantage of Bottom-Up approach:

- In this approach development and testing can be done together so that the product or application will be efficient and as per the customer specifications.

Disadvantages of Bottom-Up approach:

- We can catch the Key interface defects at the end of cycle
- It is required to create the test drivers for modules at all levels except the top control

4. **Mixed / Sandwich Integration testing:** It follows the combination of both bottom up and top down testing approaches. In top down and bottom up testing, testing can be initiated only after the modules in the same level should be coded and unit tested. The mixed approach overcomes this shortcoming of top-down and bottom up approaches.

Advantage of Mixed integration:

- Testing can be started as and when modules become available after unit testing.

SYSTEM TESTING

- In system testing the behaviour of whole system/product is tested as defined by the scope of the development project or product.
- It may include tests based on risks and/or requirement specifications, business process, use cases, or other high level descriptions of system behaviour, interactions with the operating systems, and system resources.
- System testing is most often the final test to verify that the system to be delivered meets the specification and its purpose.
- System testing is carried out by specialist testers or independent testers.
- System testing should investigate both functional and non-functional requirements of the testing.

ALPHA TESTING

- System testing done by the developer itself. This test takes place at the developer's site.
- Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.
- Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.
- Alpha testing is final testing before the software is released to the general public. It has two phases:
    - In the **first phase** of alpha testing, the software is tested by in-house developers. They use either debugger software, or hardware-assisted debuggers. The goal is to catch bugs quickly.
    - In the **second phase** of alpha testing, the software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use.

- Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

BETA TESTING

- Testing done by a friendly set of customers.
- It is also known as field testing. It takes place at **customer's site**. It sends the system to users who install it and use it under real-world working conditions.
- A beta test is the second phase of software testing in which a sampling of the intended audience tries the product out. (Beta is the second letter of the Greek alphabet.) Originally, the term *alpha test* meant the first phase of testing in a software development process. The first phase includes unit testing, component testing, and system testing. Beta testing can be considered "pre-release testing.
- The goal of beta testing is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the user's perspective that you would not want to have in your final, released version of the application.

ACCEPTANCE TESTING

- Testing done by the customer to decide whether to accept or reject the product.
- After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing.
- Acceptance testing is basically done by the user or customer although other stakeholders may be involved as well.

- The goal of acceptance testing is to establish confidence in the system.
- Acceptance testing is most often focused on a validation type testing.

REGRESSION TESTING

- ☐ During confirmation testing the defect got fixed and that part of the application started working as intended. But there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these '**unexpected side-effects**' of fixes is to do regression testing.
- ☐ The purpose of a regression testing is to verify that modifications in the software or the environment have not caused any unintended adverse side effects and that the system still meets its requirements.
- ☐ Regression testing are mostly automated because in order to fix the defect the same test is carried out again and again and it will be very tedious to do it manually.
- ☐ Regression tests are executed whenever the software changes, either as a result of fixes or new or changed functionality.

DEBUGGING

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are four main debugging strategies
  - Brute force
  - Backtracking
  - Cause elimination
  - Program slicing

BRUTE FORCE:

- Most commonly used and least efficient method
- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

BACKTRACKING

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

CAUSE ELIMINATION

- Involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- 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 cause

- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

PROGRAM SLICING

- Similar to back tracking, but the search space is reduced by defining slices.
- A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

TEST PLAN

- ☐ Test plan is the project plan for the testing work to be done. It is not a test design **specification,** a collection of **test cases** or a set of **test procedures;** in fact, most of our test plans do not address that level of detail.
- ☐ A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process.
  Master test plan: A test plan that typically addresses multiple test levels.
  Phase test plan: A test plan that typically addresses one test phase.

TEST PLAN GUIDELINES

- Make the plan concise. Avoid redundancy in your test plan.
- Be specific. For example, when you specify an operating system as a property of a test environment, mention the OS Edition/Version as well, not just the OS Name.
- Make use of lists and tables wherever possible. Avoid lengthy paragraphs.
- Have the test plan reviewed a number of times prior to baselining it or sending it for approval. The quality of your test plan speaks volumes about the quality of the testing you or your team is going to perform.
- Update the plan as and when necessary. An out-dated and unused document stinks and is worse than not having the document in the first place.

IEEE 829 STANDARD TEST PLAN TEMPLATE

Test plan identifier
Test deliverables
Introduction
Test tasks
Test items
Environmental needs
Features to be tested
Responsibilities
Features not to be tested
Staffing and training needs
Approach Schedule
Item pass/fail criteria
Risks and contingencies
Suspension and resumption criteria Approvals

TEST REPORTING

Test completion reporting is a process where test metrics are reported in summarised format to update the stakeholders which enables them to take an informed decision.

Test Completion Report Format:

- Test Summary Report Identifier
- Summary
- Variances
- Summary Results
- Evaluation
- Planned vs Actual Efforts
- Sign off

Significance of Test Completion Report:

- An indication of the quality
- Measure outstanding risks
- The level of confidence in tested software

SAMPLE TEST REPORT

TEST CASE REPORT

(Use one template for each test case)

GENERAL INFORMATION			
Test Stage:	☐ Unit ☐ Functionality ☐ Integration ☐ System ☐ Interface ☐ Performance ☐ Regression ☐ Acceptance ☐ Pilot Specify the testing stage for this test case.		
Test Date:	mm/dd/yy	System Date, if applicable:	mm/dd/yy
Tester:	Specify the name(s) of who is testing this case scenario.	Test Case Number:	Specify a unique test number assigned to the test case.
Test Case Description:	Provide a brief description of what functionality the case will test.		
Results:	☐ Pass ☐ Fail	Incident Number, if applicable:	Specify the unique identifier assigned to the incident.
INTRODUCTION			
Requirement(s) to be tested:	Identify the requirements to be tested and include the requirement number and description from the Requirements Traceability Matrix.		
Roles and Responsibilities:	Describe each project team member and stakeholder involved in the test, and identify their associated responsibility for ensuring the test is executed appropriately.		
Set Up Procedures:	Describe the sequence of actions necessary to prepare for execution of the test.		
Stop Procedures:	Describe the sequence of actions necessary to terminate the test.		
ENVIRONMENTAL NEEDS			
Hardware:	Identify the qualities and configurations of the hardware required to execute the test case.		

11. SOFTWARE QUALITY MANAGEMENT (SQM)

- The aim of Software Quality Management (SQM) is to manage the quality of software and of its development process.
- A quality product is one which meets its requirements and satisfies the user.
- A quality culture is an organizational environment where quality is viewed as everyone's responsibility.
- SQA encompasses the entire software development process, which includes processes such as requirements definition, software design, coding, source code control, code reviews, software configuration management, testing, release management, and product integration.
- SQA consists of Quality Assurance (QA), Quality control (QC) and Total Quality Management (TQM).
- Quality assurance is the process or set of processes used to measure and assure the quality of a product, whereas quality control is the process of ensuring products and services meet consumer expectations.
- Quality assurance is process oriented and focuses on defect prevention, while quality control is product oriented and focuses on defect identification.
- Total quality management (TQM) consists of organization-wide efforts to install and make permanent a climate in which an organization continuously improves its ability to deliver high-quality products and services to customers.

Difference between Quality Assurance and Quality Control

Quality Assurance	Quality Control
Quality Assurance is a part of quality management process which concentrate on providing confidence that quality requirements will be fulfilled	Quality Control is a part of quality management process which concentrates on fulfilling the quality requirements.
Quality Assurance is a set of activities for ensuring quality in the processes by which products are developed.	Quality Control is a set of activities for ensuring quality in products. The activities focus on identifying defects in the actual products produced.
Quality Assurance is the process of managing for quality;	Quality Control is used to verify the quality of the output
The goal of Quality Assurance is to prevent introducing defects in the software application which help to improve the development and testing processes.	The goal of Quality Control is to identify the defects in the software application after it is developed.
QA is Pro-active means it identifies weaknesses in the processes.	QC is Reactive means it identifies the defects and also corrects the defects or bugs also.
It does not involve executing the program or code.	It always involves executing the program or code.

Quality Assurance	Quality Control
All peoples who are involved in the developing software application as responsible for the quality assurance.	Testing team is responsible for Quality control.
Quality Assurance is process oriented	Quality Control is product oriented
Quality Assurance basically aim to prevention of defects to improve the quality.	Quality Control basically aim to detection of defects to improve the quality.
It identifies weakness in processes to improve them.	It identifies defects to be fixed.
Verification is an example of Quality Assurance.	Validation/Software Testing is an example of Quality Control.
It is a staff function.	It is a line function.
It is done before Quality Control.	It is done only after Quality Assurance activity is completed.
Quality Assurance means Planning done for doing a process.	Quality Control Means Action has taken on the process by execute them.

NEED OF SQM

- To ensure that the required level of quality is achieved in a software product.
- To encourage a company-wide "Quality Culture" where quality is viewed as everyone's responsibility.
- To reduce the learning curve and help with continuity in case team members change positions within the organization.
- To enable in-process fault avoidance and fault prevention through proper development.

SOFTWARE QUALITY FACTORS

- **Usability:** A software product should be usable for all categories of user.
- **Reusability:** Different modules of the product can be easily reused to develop new product.
- **Maintainability:** Errors in the product can be easily corrected and the functionalities of the product can be easily modified.
- **Understandability:** Understandability is possessed by a software product if the purpose of the product is clear. This goes further than just a statement of purpose - all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account, i.e. if the software product is to be used by software engineers it is not required to be understandable to the layman.
- **Completeness:** A software product possesses the characteristic **completeness** to the extent that all of its parts are present and each of its parts is fully developed. This means that if the code calls a sub-routine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must be available.

- **Conciseness:** A software product possesses the characteristic **conciseness** to the extent that no excessive information is present. This is important where memory capacity is limited, and it is important to reduce lines of code to a minimum. It can be improved by replacing repeated functionality by one sub-routine or function which achieves that functionality. It also applies to documents.
- **Portability:** A software product possesses the characteristic **portability** to the extent that it can be operated easily and well on computer configurations other than its current one. This is particularly important with PC applications where, for example, a product is expected to work on all 80486 processors.
- **Well documented:** A software product possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements. Thus the software product which is maintainable should be **well-documented,** not complex, and should have spare capacity for memory usage and processor speed.
- **Testability:** A software product possesses the characteristic **testability** to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable - a complex design leads to poor testability.
- **Convenient and practicable:** A software product possesses the characteristic usability to the extent that it is **convenient and practicable** to use. This is affected by such things as the human-computer interface. The component of the software which has most impact on this is the graphical user interface (GUI).
- **Reliability:** A software product possesses the characteristic **reliability** to the extent that it can be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whichever conditions it finds itself - this is sometimes termed robustness.

REVIEW

Review provides a powerful way to improve the quality and productivity of software development to recognize and fix their own defects early in the software development process. Nowadays, all software organizations are conducting reviews in all major aspects of their work including requirements, design, implementation, testing, and maintenance.

Advantages of Reviews

1. Types of defects that can be found during static testing are: deviations from standards, missing requirements, design defects, non-maintainable code and inconsistent interface specifications.

2. Since static testing can start early in the life cycle, early feedback on quality issues can be established, e.g. an early validation of user requirements and not just late in the life cycle during acceptance testing.

3. By detecting defects at an early stage, rework costs are relatively low and thus a relatively cheap improvement of the quality of software products can be achieved.

4. The feedback and suggestions document from the static testing process allows for process improvement, which supports the avoidance of similar errors being made in the future.

Roles and Responsibilities in a Review

There are various roles and responsibilities defined for a review process. Within a review team, four types of participants can be distinguished: moderator, author, scribe, reviewer and manager.

1. ***The moderator***:- The moderator (or review leader) leads the review process. His role is to determine the type of review, approach and the composition of the review team. The moderator also schedules the meeting, disseminates documents before the meeting, coaches other team members, paces the meeting, leads possible discussions and stores the data that is collected.

2. ***The author***:- As the writer of the 'document under review', the author's basic goal should be to learn as much as possible with regard to improving the quality of the document. The author's task is to illuminate unclear areas and to understand the defects found.

3. The scribe/ recorder:- The scribe (or recorder) has to record each defect found and any suggestions or feedback given in the meeting for process improvement.

4. The reviewer:- The role of the reviewers is to check defects and further improvements in accordance to the business specifications, standards and domain knowledge.

5. The manager :- Manager is involved in the reviews as he or she decides on the execution of reviews, allocates time in project schedules and determines whether review process objectives have been met or not.

PHASES OF FORMAL REVIEW

Formal reviews follow a formal process. It is well structured and regulated. A formal review process consists of six main steps:

1. Planning
2. Kick-off / Overview
3. Preparation
4. Review meeting
5. Rework
6. Follow-up

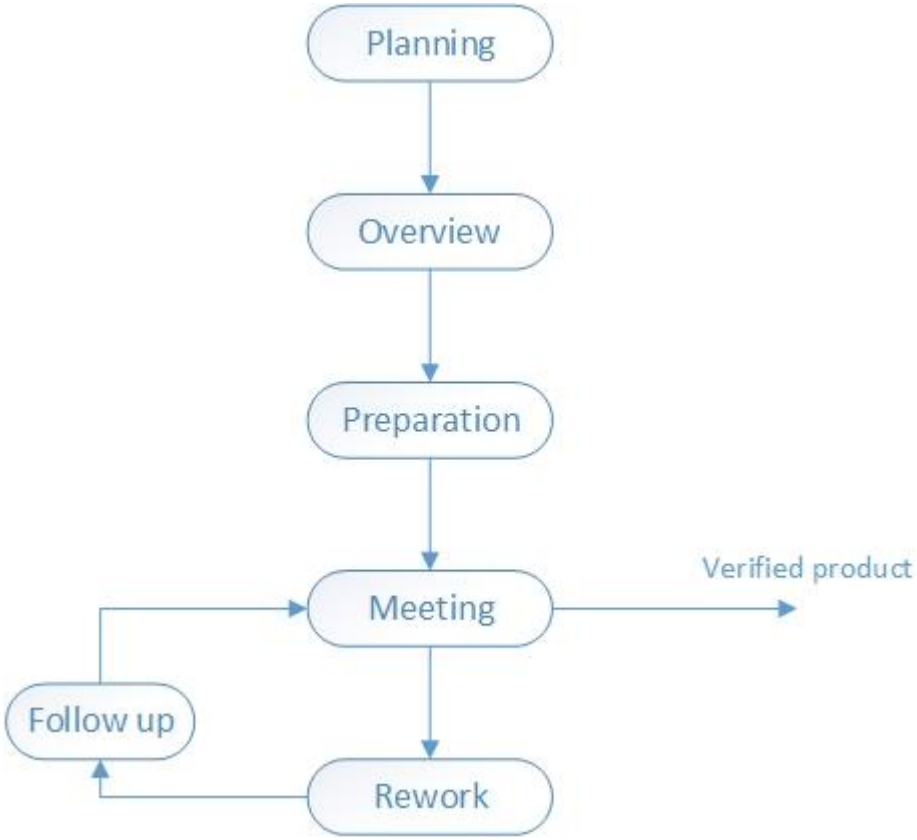


Figure: Phases of Review

1. Planning: The first phase of the formal review is the Planning phase. In this phase the review process begins with a request for review by the author to the moderator (or inspection leader). A moderator has to take care of the scheduling like date, time, place and invitation of the review. For the formal reviews the moderator performs the entry check and also defines the formal exit criteria. The **entry check** is done to ensure that the reviewer's time is not wasted on a document that is not ready for review. After doing the entry check if the document is found to have very little defects then it's ready to go for the reviews. So, the **entry criteria** are to check that whether the document is ready to enter the formal review process or not. Hence the entry criteria for any document to go for the reviews are:

- The documents should not reveal a large number of major defects.
- The documents to be reviewed should be with line numbers.
- The documents should be cleaned up by running any automated checks that apply.
- The author should feel confident about the quality of the document so that he can join the review team with that document.

Once, the document clear the entry check the moderator and author decides that which part of the document is to be reviewed. Since the human mind can understand only a limited set of pages at one time so in a review the maximum size is between 10 and 20 pages. Hence checking the documents improves the moderator ability to lead the meeting because it ensures the better understanding.

2. Kick-off / Overview: This kick-off meeting is an optional step in a review procedure. The goal of this step is to give a short introduction on the objectives of the review and the documents to everyone in the meeting. The relationships between the document under review and the other documents are also explained, especially if the numbers of related documents are high.

3. Preparation: In this step the reviewers review the document individually using the related documents, procedures, rules and checklists provided. Each participant while reviewing individually identifies the defects, questions and comments according to their understanding of the document and role. After that all issues are recorded using a logging form. The success factor for a thorough preparation is the number of pages checked per hour. This is called the **checking rate.** Usually the checking rate is in the range of 5 to 10 pages per hour.

4. Review meeting: The review meeting consists of three phases:

Logging phase: In this phase the issues and the defects that have been identified during the preparation step are logged page by page. The logging is basically done by the author or by a **scribe.** Scribe is a separate person to do the logging and is especially useful for the formal review types such as an inspection. Every defects and its severity should be logged in any of the three severity classes given below

- ✓ **Critical:** The defects **will cause** downstream damage.
- ✓ **Major:** The defects **could cause** a downstream damage.
- ✓ **Minor:** The defects are **highly unlikely to cause** the downstream damage.

During the logging phase the moderator focuses on logging as many defects as possible within a certain time frame and tries to keep a good logging rate (number of defects logged per minute). In formal review meeting the good logging rate should be between one and two defects logged per minute.

- **Discussion phase:** If any issue needs discussion then the item is logged and then handled in the discussion phase. As chairman of the discussion meeting, the moderator takes care of the people issues and prevents discussion from getting too personal and calls for a break to cool down the heated discussion. The outcome of the discussions is documented for the future reference.

- **Decision phase:** At the end of the meeting a decision on the document under review has to be made by the participants, sometimes based on formal **exit criteria. Exit criteria** are the average number of critical and/or major defects found per page (for example no more than three critical/major defects per page). If the number of defects found per page is more than a certain level then the document must be reviewed again, after it has been reworked.

5. Rework: In this step if the number of defects found per page exceeds the certain level then the document has to be reworked. Not every defect that is found leads to rework. It is the author's responsibility to judge whether the defect has to be fixed. If nothing can be done about an issue then at least it should be indicated that the author has considered the issue.

6. Follow-up: In this step the moderator check to make sure that the author has taken action on all known defects. If it is decided that all participants will check the updated documents then the moderator takes care of the distribution and collects the feedback. It is the responsibility of the moderator to ensure that the information is correct and stored for future analysis.

TYPES OF REVIEW

1. Walkthrough:

- It is not a formal process
- It is led by the authors
- Author guide the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.

- Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- Is especially useful for higher level documents like requirement specification, etc.

The goals of a walkthrough:

- To present the documents both within and outside the software discipline in order to gather the information regarding the topic under documentation.
- To explain or do the knowledge transfer and evaluate the contents of the document
- To achieve a common understanding and to gather feedback.
- To examine and discuss the validity of the proposed solutions

2. Technical review:

- It is less formal review
- It is led by the trained moderator but can also be led by a technical expert
- It is often performed as a peer review without management participation
- Defects are found by the experts (such as architects, designers, key users) who focus on the content of the document.
- In practice, technical reviews vary from quite informal to very formal

The goals of the technical review are:

- To ensure that an early stage the technical concepts are used correctly
- To access the value of technical concepts and alternatives in the product
- To have consistency in the use and representation of technical concepts
- To inform participants about the technical content of the document

3. Inspection:

- It is the most formal review type
- It is led by the trained moderators
- During inspection the documents are prepared and checked thoroughly by the reviewers before the meeting
- It involves peers to examine the product
- A separate preparation is carried out during which the product is examined and the defects are found
- The defects found are documented in a logging list or issue log
- A formal follow-up is carried out by the moderator applying exit criteria

The goals of inspection are:

- It helps the author to improve the quality of the document under inspection
- It removes defects efficiently and as early as possible
- It improve product quality
- It create common understanding by exchanging information
- It learn from defects found and prevent the occurrence of similar defects

SOFTWARE CONFIGURATION MANAGEMENT (SCM)

- Software Configuration Management is the ability to control and manage change in a software project.
- Change is inherent and ongoing in any software project. The ability to track control such changes in a proper manner form the basis of a good software project. Software Configuration Management tries to bridge this gap by defining a process for change control.
- SCM is the process that defines how to control and manage change.
- Change Management defines processes to prevent unauthorized changes, procedures to follow when making changes, required information, possibly workflow management as well. Change management is orders of magnitude more complex than version control of software.

Need of SCM / SCM Process Objectives

1. Identify all items that define the software configuration
2. Manage changes to one or more configuration items
3. Facilitate construction of different versions of a software application
4. Ensure that software quality is maintained as configuration evolves

Software Configuration Management Tasks

- Identification (tracking multiple versions to enable efficient changes)
- Version control (control changes before and after release to customer)
- Change control (authority to approve and prioritize changes)
- Configuration auditing (ensure changes made properly)
- Reporting (tell others about changes made)

Configuration Object Identification

- To control and manage configuration items, each must be named and managed using an object-oriented approach
- Basic objects are created by software engineers during analysis, design, coding, or testing
- Aggregate objects are collections of basic objects and other aggregate objects
- Configuration object attributes: unique name, description, list of resources, and a realization (a pointer to a work product for a basic object or null for an aggregate object)

Version Control

- Combines procedures and tools to manage the different versions of configuration objects created during the software process
- Version control systems require the following capabilities
  - Project repository – stores all relevant configuration objects
  - Version management capability – stores all versions of a configuration object (enables any version to be built from past versions)
  - Make facility – enables collection of all relevant configuration objects and construct a specific software version
  - Issues (bug) tracking capability – enables team to record and track status of outstanding issues for each configuration object
- Uses a system modelling approach (template – includes component hierarchy and component build order, construction rules, verification rules)

Change Control

- Change request is submitted and evaluated to assess technical merit and impact on the other configuration objects and budget
- Change report contains the results of the evaluation
- Change control authority (CCA) makes the final decision on the status and priority of the change based on the change report
- Engineering change order (ECO) is generated for each change approved (describes change, lists the constraints, and criteria for review and audit)
- Object to be changed is checked-out of the project database subject to access control parameters for the object
- Modified object is subjected to appropriate SQA and testing procedures
- Modified object is checked-in to the project database and version control mechanisms are used to create the next version of the software
- Synchronization control is used to ensure that parallel changes made by different people don't overwrite one another

SCM PROCESS

- ❖ Traditional SCM process is looked upon as the best fit solution to handling changes in software projects. Traditional SCM process identifies the functional and physical attributes of a software at various points in time and performs systematic control of changes to the identified attributes for the purpose of maintaining software integrity and traceability throughout the software development life cycle.
- ❖ The SCM process further defines the need to trace the changes and the ability to verify that the final delivered software has all the planned enhancements that are supposed to be part of the release.
- ❖ The traditional SCM identifies four procedures that must be defined for each software project to ensure a good SCM process is implemented.

i. Configuration Identification
ii. Configuration Control
iii. Configuration Status Accounting
iv. Configuration Authentication

1. Configuration Identification

- ✓ Software is usually made up of several programs. Each program, its related documentation and data can be called as a "configurable item"(CI). The number of CI in any software project and the grouping of artefacts that make up a CI is a decision made of the project. The end product is made up of a bunch of CIs.
- ✓ The status of the CIs at a given point in time is called as a *baseline*. The baseline serves as a reference point in the software development life cycle. Each new baseline is the sum total of an older baseline plus a series of approved changes made on the CI
- ✓ A baseline is considered to have the following attributes

  - ➢ **Functionally complete:** A baseline will have a defined functionality. The features and functions of this particular baseline will be documented and available for reference. Thus the capabilities of the software at a particular baseline is well known.

- **Known Quality:** The quality of a baseline will be well defined. i.e. all known bugs will be documented and the software will have undergone a complete round of testing before being put define as the baseline.
- **Immutable and completely re-creatable:** A baseline, once defined, cannot be changed. The list of the CIs and their versions are set in stone. Also, all the CIs will be under version control so the baseline can be recreated at any point in time.

2. Configuration Control

- ✓ The process of deciding, co-ordinating the approved changes for the proposed CIs and implementing the changes on the appropriate baseline is called Configuration control.
- ✓ It should be kept in mind that configuration control only addresses the process after changes are approved. The act of evaluating and approving changes to software comes under the purview of an entirely different process called change control.

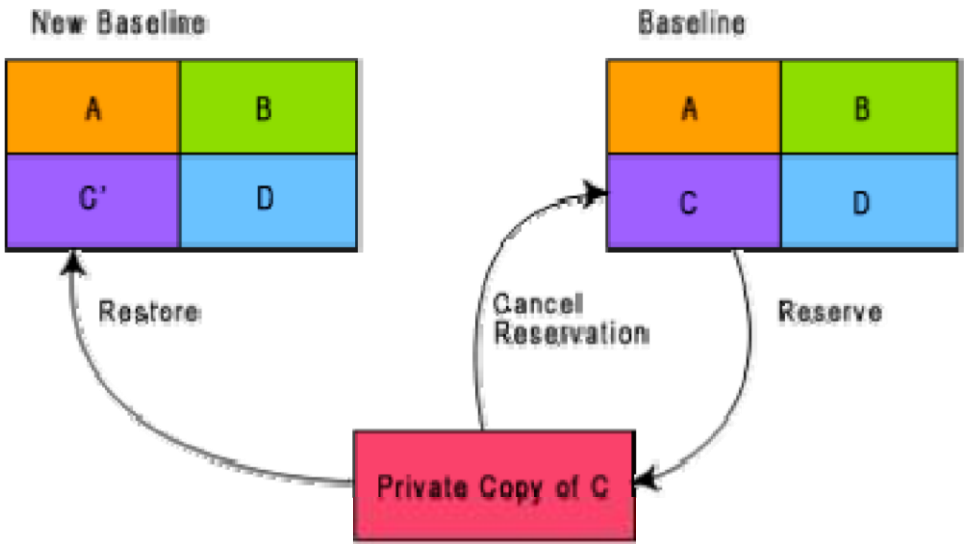


Figure: Reserve and restore operation in configuration control

It can be shown how the changes to any object that is under configuration control can be achieved. The engineer needing to change a module first obtains a private copy of the module through a reserve operation. Then, he carries out all necessary changes on this private copy. However, restoring the changed module to the system configuration requires the permission of a change control board (CCB). The CCB is usually constituted from among the development team members. For every change that needs to be carried out, the CCB reviews the changes made to the controlled object and certifies several things about the change:

1. Change is well-motivated.
2. Developer has considered and documented the effects of the change.
3. Changes interact well with the changes made by other developers.
4. Appropriate people (CCB) have validated the change, e.g. someone has tested the changed code, and has verified that the change is consistent with the requirement.

The change control board (CCB) sounds like a group of people. However, except for very large projects, the functions of the change control board are normally discharged by the project manager himself or some senior member of the development team. Once the CCB reviews the changes to the module, the project manager updates the old base line through a restore operation. A configuration control tool does not allow a developer to replace an object he has reserved with his local copy unless he gets an authorization from the CCB. By constraining the developers' ability to replace reserved objects, a stable environment is achieved. Since a configuration management tool allows only one engineer to work on one module at any one time, problem of accidental overwriting is eliminated. Also, since only the manager can update the baseline after the CCB approval, unintentional changes are eliminated.

3. Configuration Status Accounting

- ✓ Configuration status accounting is the bookkeeping process of each release. This procedure involves tracking what is in each version of software and the changes that lead to this version.
- ✓ Configuration status accounting keeps a record of all the changes made to the previous baseline to reach the new baseline.

4. Configuration Authentication

- ✓ Configuration authentication (CA) is the process of assuring that the new baseline has all the planned and approved changes incorporated. The process involves verifying that all the functional aspects of the software is complete and also the completeness of the delivery in terms of the right programs, documentation and data are being delivered.
- ✓ The configuration authentication is an audit performed on the delivery before it is opened to the entire world.

ADVANTAGES OF SCM

- Reduced redundant work.
- Effective management of simultaneous updates.
- Avoids configuration-related problems.
- Facilitates team coordination.
- Helps in building management; managing tools used in builds.
- Defect tracking: It ensures that every defect has traceability back to its source.

SOFTWARE RELIABILITY

- Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.
- It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced.
- Reliability of a product depends not only on the number of latent errors but also on the exact location of the errors.
- Reliability also depends upon how the product is used, i.e. on its execution profile.

Reasons for software reliability being difficult to measure

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

RELIABILITY METRICS

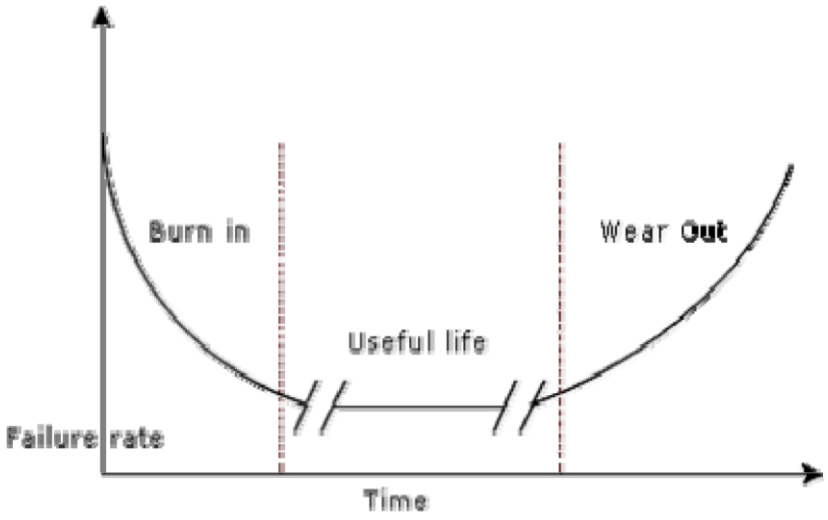
- The reliability requirements for different categories of software products may be different.
- Some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. Various reliability metrics are listed below.

  - **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behaviour (i.e. failures). ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.
    - **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures
  - **Mean Time To Repair (MTTR).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
  - **Mean Time Between Failure (MTBR).** MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.
  - **Probability of Failure on Demand (POFOD).** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.
  - **Availability.** Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

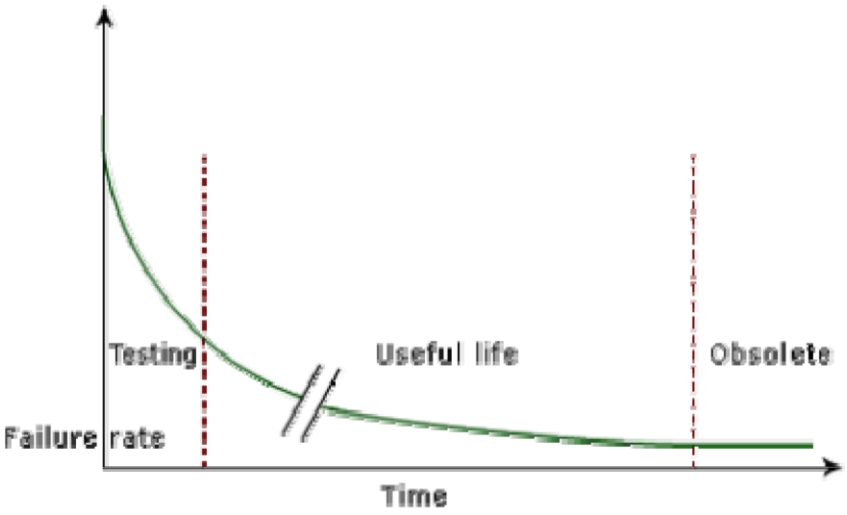
Hardware reliability vs. Software reliability

- Reliability behaviour for hardware and software are very different.
- For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is

concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase).



(a) Hardware product



(b) Software product

Figure: Change in failure rate of a product

- The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 13.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics "bath tub" shape. On the other hand, for software the failure rate is at its highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.

12. ISO 9000 STANDARD

- ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987. ISO certification serves as a reference for contract between independent parties.
- The ISO 9000 standard specifies the guidelines for maintaining a quality system. The ISO standard mainly addresses operational aspects and organizational aspects such as responsibilities, reporting, etc.
- In a nutshell, ISO 9000 specifies a set of guidelines for repeatable and high quality product development. ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.

- **OVERVIEW OF ISO 9000 SERIES**

ISO Standard	**Title**	**Description**
ISO 9000	*Quality Management and Quality Assurance Standards--Guidlines for Selection and Use*	Guidelines for the selection and use of ISO 9001, 9002 and 9003.
ISO 9001*	*Quality Systems--Model for quality assurance in design/development, production, installation and servicing.*	Standard covers design, development, production, installation, and servicing, **this applies to the software industry.**
ISO 9002	*Quality systems--Model for quality assurance in production and installation.*	Assesses the production and installation processes.
ISO 9003	*Quality systems--Model for quality assurance in final inspection and test.*	Evaluation the final inspection and test phase.
ISO 9004	*Quality management and quality system elements—Guidelines*	Defines the 20 fundamental quality system concepts included in the three models.

Need for obtaining ISO 9000 certification / Why to get ISO certification

- Confidence of customers in an organization increases when organization qualifies for ISO certification. This is especially true in the international market. In fact, many organizations awarding international software development contracts insist that the development organization have ISO 9000 certification. For this reason, it is vital for software organizations involved in software export to obtain ISO 9000 certification.
- ISO 9000 requires a well-documented software production process to be in place. A well-documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient, and cost effective.
- ISO 9000 certification points out the weak points of an organization and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and Total Quality Management (TQM).

HOW TO GET ISO 9000 CERTIFICATION

1. **Application stage:** The organisation should apply for ISO certification to registrar for registration
2. **Pre –assessment:** During this stage the registrar makes a rough assessment of the organisation.
3. **Document review & adequacy audit:** The registrar reviews the documents submitted by the organisation and makes suggestions for possible improvements.
4. **Compliance audit:** The registrar checks whether the suggestions made by it during review have been complied to by the organisation or not.
5. **Registration:** The registrar awards the ISO 9000 certificate after successful completion of all previous phases.
6. **Continued surveillance:** The registrar continues monitoring the organisation periodically.

Summary of ISO 9001 certification

A summary of the main requirements of ISO 9001 as they relate of software development is as follows:

Management Responsibility (4.1)

- The management must have an effective quality policy.
- The responsibility and authority of all those whose work affects quality must be defined and documented.
- A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
- The effectiveness of the quality system must be periodically reviewed by audits.

Quality System (4.2)

A quality system must be maintained and documented.

Contract Reviews (4.3)

Before entering into a contract, an organization must review the contract to ensure that it is understood, and that the organization has the necessary capability for carrying out its obligations.

Design Control (4.4)

- The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
- Design inputs must be verified as adequate.
- Design must be verified.
- Design output must be of required quality.
- Design changes must be controlled.

Document Control (4.5)

- There must be proper procedures for document approval, issue and removal.
- Document changes must be controlled. Thus, use of some configuration management tools is necessary.

Purchasing (4.6)
Purchasing material, including bought-in software must be checked for conforming to requirements.

Purchaser Supplied Product (4.7)
Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product Identification (4.8)
The product must be identifiable at all stages of the process. In software terms this means configuration management.

Process Control (4.9)
- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and Testing (4.10)
In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

Inspection, Measuring and Test Equipment (4.11)
If integration, measuring, and test equipment are used, they must be properly maintained and calibrated.

Inspection and Test Status (4.12)
The status of an item must be identified. In software terms this implies configuration management and release control.

Control of Nonconforming Product (4.13)
In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

Corrective Action (4.14)
This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the system needs improvement.

Handling, (4.15)
This clause deals with the storage, packing, and delivery of the software product.

Quality records (4.16)
Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

Quality Audits (4.17)
Audits of the quality system must be carried out to ensure that it is effective.

Training (4.18)
Training needs must be identified and met.

FEATURES OF ISO 9001 CERTIFICATION

The salient features of ISO 9001 are as follows:

- **Document control:** All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.
- **Planning:** Proper plans should be prepared and then progress against these plans should be monitored.
- **Review:** Important documents should be independently checked and reviewed for effectiveness and correctness.
- **Testing:** The product should be tested against specification.
- **Organizational aspects:** Several organizational aspects should be addressed e.g., management reporting of the quality team.

SHORTCOMINGS OF ISO 9000

- ISO 9000 requires a software production process to be adhered to but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
- ISO 9000 certification process is not fool-proof and no international accreditation agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accreditation agencies and also among the registrars.
- Organizations getting ISO 9000 certification often tend to downplay domain expertise. These organizations start to believe that since a good process is in place, any engineer is as effective as any other engineer in doing any particular activity relating to software development. However, many areas of software development are so specialized that special expertise and experience in these areas (domain expertise) is required. In manufacturing industry there is a clear link between process quality and product quality. Once a process is calibrated, it can be run again and again producing quality goods. In contrast, software development is a creative process and individual skills and experience are important.
- ISO 9000 does not automatically lead to continuous process improvement, i.e. does not automatically lead to TQM.

ISO 9000-3 Model

ISO 9000-3 divides the elements of its quality system model into three parts: a framework, the supporting structures, and the life cycle activities. The terminology of 9000-3 is software development-oriented, rather than manufacturing-oriented. Nevertheless, all of 9001's requirements are covered by 9000-3.

ISO 9000-3, Primary Areas of Concern

The ISO 9000 Approach to Quality Systems for Software Development, describes some of the primary areas of concern in 9000-3. These include configuration management, change control, development planning, quality planning, design and implementation, testing and validation, and maintenance.

- **Configuration Management**

The purpose of configuration management is to ensure each build of a product is derived from the correct version of every source file. These requirements apply to all software products, including customer documentation and manuals. A configuration management plan should identify the activities to be carried out and the tools, techniques, and methods to be used.

The plan should determine the stage at which items will be brought under configuration control. Each software item should be identified uniquely and each item's build status should be identified. The version of each software item required to construct each version of the product should be specified.

- **Change Control**

Change control helps managers balance customer needs, technical capability, and additional resource requirements for modifications to the product. It helps prevent the late introduction of new requirements in the development cycle Change control also ensures all parties affected by a change are notified. Procedures should be developed to identify, document, review, and authorize any changes to items under configuration management.

It should be possible to trace approved changes and subsequent modifications to all affected items, from change initiation through release. Changes are often initiated by engineering change orders (ECOs) also known as modification requests, initiation requests, etc. A review board typically reviews the ECOs, prioritizes them, and decides whether to accept immediately, defer, or reject the requests based on previously established criteria.

- **Development Planning**

A development plan provides a unified view of the activities required to complete a project. It includes identification of the activities to be performed, the resources required for each activity, and the timing of and coordination of each activity. Risk analysis and contingency planning should be included as part of the development planning process.

A development plan should define the development stages to be carried out for each project, the required inputs and outputs for each phase, and the verification procedures for each phase. The development plan should include a schedule, describe the mechanisms for tracking progress, and outline organizational responsibilities and work assignments. The development plan should identify related project plans such as the quality plan, integration plan, test plan, maintenance plan, and configuration management plan.

The development plan provides the basis for tracking project progress. Completion of milestones should be recorded and progress should be compared to the development plan schedule on a regular basis. Slippages should be identified and the plan should be updated to accommodate additional resources required, including time, personnel, and equipment. If the shipping date will be changed, affected departments and customers should be notified so they may plan accordingly.

- **Quality Planning**

The purpose of a quality plan is to determine a project's quality goals before development starts. This information helps direct the testing and validation effort and determines whether or not a product has achieved

its quality goals and, therefore, is ready to be shipped to customers. In the absence of a quality plan, products are often shipped at the schedule's end date without regard to product quality.

A quality plan should state the project quality goals–in measurable terms, when possible. It should define the input and output criteria for each development phase and identify the test and verification activities to be carried out for each phase. The plan should also identify those who are responsible for and have authority for the quality assurance activities performed.

- **Design and Implementation**

The reasons for using a design and implementation methodology include use of common language to express and evaluate designs, consistent approach among projects, and elimination of haphazard development. Projects of different size and complexity may elect to use different design methods and degrees of formality.

However, ISO 9000-3 suggests that a systematic design methodology appropriate for the software product being developed should be identified and used. Design and coding rules and conventions should be identified. Reviews should be carried out to ensure the product requirements are met and the methods identified are employed appropriately.

- **Testing and Validation**

A test plan identifies the levels and types of tests to be run on a product, the resources required, the schedule, and the required inputs and expected outputs for each test case. 9000-3 recognizes that several types of testing may be necessary to adequately exercise a product, such as unit, integration, system, and acceptance testing.

The test plan should include descriptions of test environments, tools, software, and documentation needed. The required version of each software and hardware component in the test environment should be specified. The inputs and expected results for each test case should be documented. Completion criteria for each level of testing should be described.

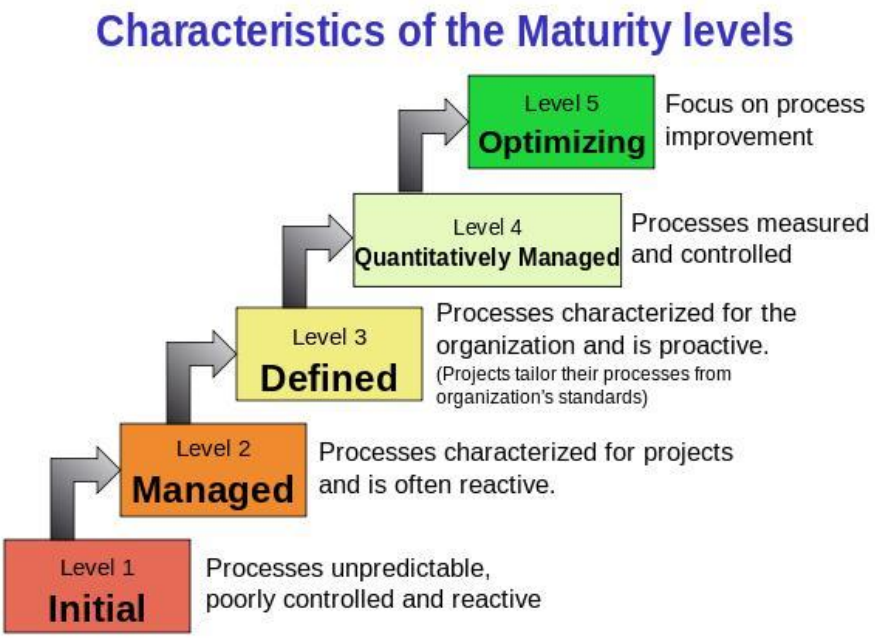
- **Maintenance**

Because the maintenance phase of a product is likely to last much longer than the original product design and development, this phase should consist of a carefully planned set of activities. A maintenance plan should identify support organizations, activities covered under the plan, the records and reports produced, and release procedures for distributing and installing customer updates. The maintenance plan should cover problem resolution, interface modifications, and functional enhancement for the product.

CAPABILITY MATURITY MODEL (CMM)

- Capability Maturity Model is a bench-mark for measuring the maturity of an organization's software process. It is a methodology used to develop and refine an organization's software development process. CMM can be used to assess an organization against a scale of five process maturity levels based on certain Key Process Areas (KPA). It describes the maturity of the company based upon the project the company is dealing with and the clients. Each level ranks the organization according to its standardization of processes in the subject area being assessed.
- A maturity model provides:
    - A place to start
    - The benefit of a community's prior experiences
    - A common language and a shared vision
    - A framework for prioritizing actions
    - A way to define what improvement means for your organization
- In CMMI models with a staged representation, there are five maturity levels designated by the numbers 1 through 5 as shown below:

1. Initial
2. Repeatable / Managed
3. Defined
4. Quantitatively Managed
5. Optimizing



- Maturity levels consist of a predefined set of process areas. The maturity levels are measured by the achievement of the **specific** and **generic goals** that apply to each predefined set of process areas. The following sections describe the characteristics of each maturity level in detail.
- Key Process Areas (KPAs) of each level are shown in below table.

CMM Level	Focus	Key Process Ares
1. Initial	Competent people	
2. Repeatable	Project management	Software project planning Software configuration management
3. Defined	Definition of processes	Process definition Training program Peer reviews
4. Managed	Product and process quality	Quantitative process metrics Software quality management
5. Optimizing	Continuous process improvement	Defect prevention Process change management Technology change management

Maturity Level 1 – Initial:

Company has no standard process for software development. Nor does it have a project-tracking system that enables developers to predict costs or finish dates with any accuracy.

- At maturity level 1, processes are usually ad hoc and chaotic.
- The organization usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes.
- Maturity level 1 organizations often produce products and services that work but company has no standard process for software development. Nor does it have a project-tracking system that enables developers to predict costs or finish dates with any accuracy.
- Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.

Maturity Level 2 – Repeatable / Managed:

Company has installed basic software management processes and controls. But there is no consistency or coordination among different groups.

- At maturity level 2, an organization has achieved all the **specific** and **generic goals** of the maturity level 2 process areas. In other words, the projects of the organization have ensured that requirements are managed and that processes are planned, performed, measured, and controlled.
- The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.
- At maturity level 2, requirements, processes, work products, and services are managed. The status of the work products and the delivery of services are visible to management at defined points.
- Commitments are established among relevant stakeholders and are revised as needed. Work products are reviewed with stakeholders and are controlled.
- The work products and services satisfy their specified requirements, standards, and objectives.

Maturity Level 3 – Defined:

Company has pulled together a standard set of processes and controls for the entire organization so that developers can move between projects more easily and customers can begin to get consistency from different groups.

- At maturity level 3, an organization has achieved all the **specific** and **generic goals.**
- At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods.
- A critical distinction between maturity level 2 and maturity level 3 is the scope of standards, process descriptions, and procedures. At maturity level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At maturity level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit.
- The organization's set of standard processes includes the processes addressed at maturity level 2 and maturity level 3. As a result, the processes that are performed across the organization are consistent except for the differences allowed by the tailoring guidelines.
- Another critical distinction is that at maturity level 3, processes are typically described in more detail and more rigorously than at maturity level 2.
- At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.
- ISO 9000 aims at achieving this level.

Maturity Level 4 – Managed / Quantitatively Managed:

In addition to implementing standard processes, company has installed systems to measure the quality of those processes across all projects.

- At maturity level 4, an organization has achieved all the **specific goals** of the process areas assigned to maturity levels 2, 3, and 4 and the **generic goals** assigned to maturity levels 2 and 3.
- At maturity level 4 Sub-processes are selected that significantly contribute to overall process performance. These selected sub-processes are controlled using statistical and other quantitative techniques.
- Quantitative objectives for quality and process performance are established and used as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance are understood in statistical terms and are managed throughout the life of the processes.
- For these processes, detailed measures of process performance are collected and statistically analysed. Special causes of process variation are identified and, where appropriate, the sources of special causes are corrected to prevent future occurrences.
- Quality and process performance measures are incorporated into the organizations measurement repository to support fact-based decision making in the future.
- A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.

Maturity Level 5 – Optimizing:

Company has accomplished all of the above and can now begin to see patterns in performance over time, so it can tweak its processes in order to improve productivity and reduce defects in software development across the entire organization.

- At maturity level 5, an organization has achieved all the **specific goals** of the process areas assigned to maturity levels 2, 3, 4, and 5 and the **generic goals** assigned to maturity levels 2 and 3.
- Processes are continually improved based on a quantitative understanding of the common causes of variation inherent in processes.
- Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements.
- Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement.
- The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.
- Optimizing processes that are agile and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization.
- The organization's ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning. Improvement of the processes is inherently part of everybody's role, resulting in a cycle of continual improvement.
- A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process (that is, shifting the mean of the process performance) to improve process performance (while maintaining statistical predictability) to achieve the established quantitative process-improvement objectives.

Shortcomings of CMM

1. Need more guidance to the organisation to improve their quality.
2. The organisation should maintain thicker documents and longer meetings.
3. Getting the accurate measure of an organisation's current maturity level is not feasible.

SUMMARY OF CMM

Maturity Level	Rating	Description	KPAs...
5	Optimizing	Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.	cover the issues that both the organization and the process must address to implement continual, measurable software process improvement. The KPAs are: • Defect Prevention • Technology Change Management • Process Change Management.
4	Managed	Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.	focus on establishing a quantitative understanding of both the software process and the software work products being built. The KPAs are: • Quantitative Process Management • Software Quality Management
3	Defined	The software process for both management and engineering activities is documented, standardized and integrated into standard software processes for the organization. All project use an approved, tailored version of the organization's standard software process for developing and maintaining software.	address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects. The KPAs are: • Organization Process Focus • Organization Process Definition • Training Program • Integrated Software Management • Software Product Engineering • Intragroup Coordination • Peer Reviews
2	Repeatable	Basic project management processes are established to track cost, schedule, and functionality. The	Focus on the software project's concerns related to establishing

		necessary process discipline is in place to repeat earlier successes on project with similar applications.	basic project management controls. The KPAs are: • Requirements Management • Software Project Planning • Software Project Tracking and Oversight • Software Subcontract Management • Software Quality Assurance • Software Configuration Management
1	Initial	The software process is characterized as ad-hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.	None

ISO 9000 vs SEI/CMM

The characteristics of ISO 9000 certification and the SEI CMM differ in some respects. The differences are as follows:

- ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organization in official documents, communication with external parties, and the tender quotations. However, SEI CMM assessment is purely for internal use.
- SEI CMM was developed specifically for software industry and therefore addresses many issues which are specific to software industry alone.
- SEI CMM goes beyond quality assurance and prepares an organization to ultimately achieve Total Quality Management (TQM). In fact, ISO 9001 aims at level 3 of SEI CMM model.
- SEI CMM model provides a list of key process areas (KPAs) on which an organization at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement.

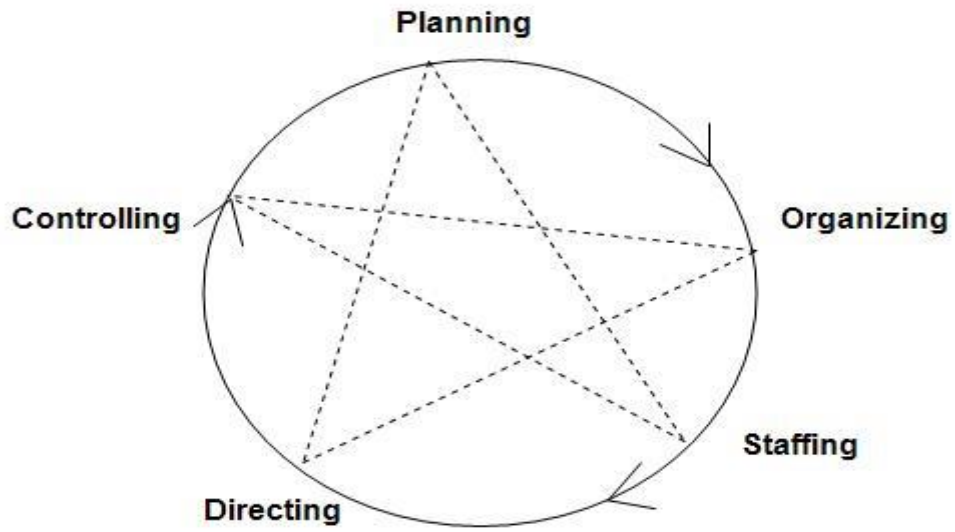
➢ **COMPARISON BETWEEN ISO & CMM**

ISO	**CMM**
Minimum requirements with implied continuous improvement	Explicit Continuous Quality Improvement
Not specific to any one industry or service	Software specific
Outwardly focused from the firm	Inwardly focused to the firm
Registration Document	No Documentation
Continual Audits	No follow up audits

13. SOFTWARE PROJECT MANAGEMENT

A system of management procedures, practices, technologies, skills, and experience necessary to successfully manage a software project

- The main goal of software project management is to enable a group of software developers to work efficiently towards successful completion of the project.
- The entire SPM consist of five phases: Planning, Staffing, Organizing, Controlling & Directing.



PLANNING

- ✓ It is the basic function of management. It deals with chalking out a future course of action & deciding in advance the most appropriate course of actions for achievement of pre-determined goals.
- ✓ Planning is deciding in advance - what to do, when to do & how to do. It bridges the gap from where we are & where we want to be".
- ✓ A plan is a future course of actions. It is an exercise in problem solving & decision making.
- ✓ Planning is determination of courses of action to achieve desired goals. Thus, planning is a systematic thinking about ways & means for accomplishment of pre-determined goals.

Planning activities

- Set objectives and goals
- Develop strategies
- Develop policies
- Forecast future situations
- Conduct a risk assessment
- Determine possible courses of action
- Make planning decisions
- Set procedures and rules
- Develop project plans
- Prepare budgets
- Document project plans

ORGANIZING

- ✓ It is the process of bringing together physical, financial and human resources and developing productive relationship amongst them for achievement of organizational goals.
- ✓ To organize a business is to provide it with everything useful or its functioning i.e. raw material, tools, capital and personnel's.
- ✓ To organize a business involves determining & providing human and non-human resources to the organizational structure.

Organizing activities

- *Identify and group project function, activities, and tasks*
- *Select organizational structures*
- Create organizational positions
- Define responsibilities and authority
- Establish position qualifications
- Document organizational decisions

STAFFING

- ✓ It is the function of manning the organization structure and keeping it manned.
- ✓ Staffing has assumed greater importance in the recent years due to advancement of technology, increase in size of business, complexity of human behavior etc.
- ✓ The main purpose of staffing is to put right man on right job.
- ✓ Managerial function of staffing involves manning the organization structure through proper and effective selection, appraisal & development of personnel to fill the roles designed under the structure.

Staffing activities

- Fill organizational positions
- Assimilate newly assigned personnel
- Educate or train personnel
- Provide for general development
- Evaluate and appraise personnel
- Compensate
- Terminate assignments
- Document staffing decisions

DIRECTING

- ✓ It is that part of managerial function which actuates the organizational methods to work efficiently for achievement of organizational purposes.
- ✓ It is considered life-spark of the enterprise which sets it in motion the action of people because planning, organizing and staffing are the mere preparations for doing the work.
- ✓ Direction is that inert-personnel aspect of management which deals directly with influencing, guiding, supervising, motivating sub-ordinate for the achievement of organizational goals.
- ✓ Direction has following elements:

Supervision- implies overseeing the work of subordinates by their superiors. It is the act of watching & directing work & workers.

Motivation- means inspiring, stimulating or encouraging the sub-ordinates with zeal to work. Positive, negative, monetary, non-monetary incentives may be used for this purpose.

Leadership- may be defined as a process by which manager guides and influences the work of subordinates in desired direction.

Communications- is the process of passing information, experience, opinion etc from one person to another. It is a bridge of understanding.

Directing activities

- *Provide leadership*
- *Supervise personnel*
- *Delegate authority*
- *Motivate personnel*
- Build teams
- Coordinate activities
- Facilitate communication
- Resolve conflicts
- Manage changes
- Document directing decisions

CONTROLLING

- ✓ It implies measurement of accomplishment against the standards and correction of deviation if any to ensure achievement of organizational goals.
- ✓ The purpose of controlling is to ensure that everything occurs in conformities with the standards. An efficient system of control helps to predict deviations before they actually occur.
- ✓ Controlling is the process of checking whether or not proper progress is being made towards the objectives and goals and acting if necessary, to correct any deviation
- ✓ Controlling is the measurement & correction of performance activities of subordinates in order to make sure that the enterprise objectives and plans desired to obtain them as being accomplished.

Controlling activities

- Develop standards of performance
- Establish monitoring and reporting systems
- Measure and analyze results
- Initiate corrective actions
- Reward and discipline
- Document controlling methods

ORGANIZATION STRUCTURES

- ✓ Usually every software development organization handles several projects at any time. Software organizations assign different teams of engineers to handle different software projects.
- ✓ There are essentially two broad ways in which a software development organization can be structured: functional format and project format.

PROJECT FORMAT

- ➢ In the project format, the project development staff are divided based on the project for which they work as shown below.

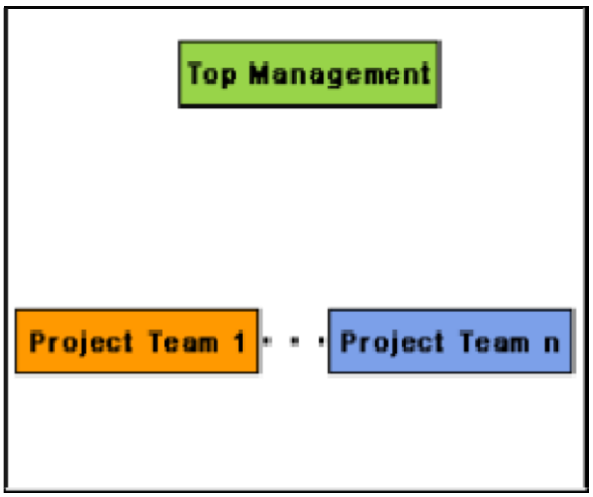


Figure: Project Organization

- ➢ In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities.
- ➢ A project organization structure forces the manager to take in almost a constant number of engineers for the entire duration of his project. This results in engineers idling in the initial phase of the software development and are under tremendous pressure in the later phase of the development.
- ➢ The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, coder, tester, etc during the course of the project. So a team member can fill any slots in the development team.
- ➢ In project format, each individual team consist of a representative to do various SDLC phases like analysis, design, coding, testing, architecture design, reviewer etc.
- ➢ Though each member is aware about various SDLC activities, it is easy to rotate the roles of team members.
- ➢ The team members can improve their knowledge and skills in every domains of SDLC.

FUNCTIONAL FORMAT

- In the functional format, the development staff are divided based on the functional group to which they belong. The different projects borrow engineers from the required functional groups for specific phases to be undertaken in the project and return them to the functional group upon the completion of the phase.

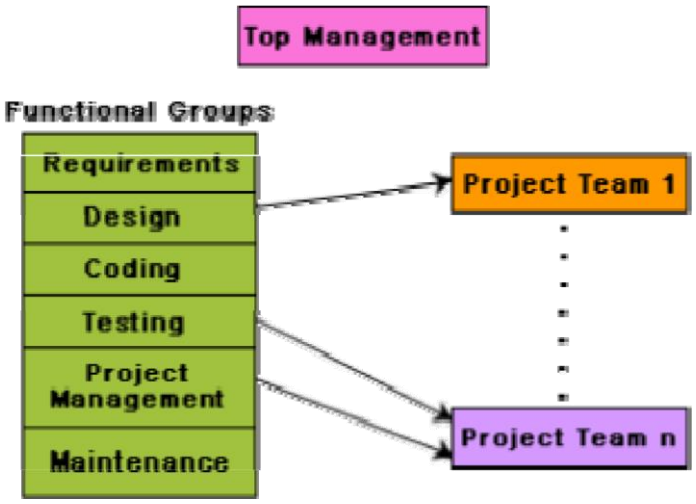


Figure: Functional Organization

- In the functional format, different teams of programmers perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the project evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood by the subsequent teams working on the project. This requires good quality documentation to be produced after every activity.
- The main advantages of a functional organization are:

  - Ease of staffing
  - Production of good quality documents
  - Job specialization
  - Efficient handling of the problems associated with manpower turnover.

- The functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams.
- The functional organization allows the engineers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work.

- It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organization also provides an efficient solution to the staffing problem.
- The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organization.
- Functional format is not suitable for small organizations handling just one or two projects.
- Another problem with the functional organization is that if an organization handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects.

TEAM STRUCTURE

- Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized.
- There are mainly three formal team structures: chief programmer, democratic, and the mixed team organization.

1. Chief Programmer Team

- In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members.
- The structure of the chief programmer team is shown below.

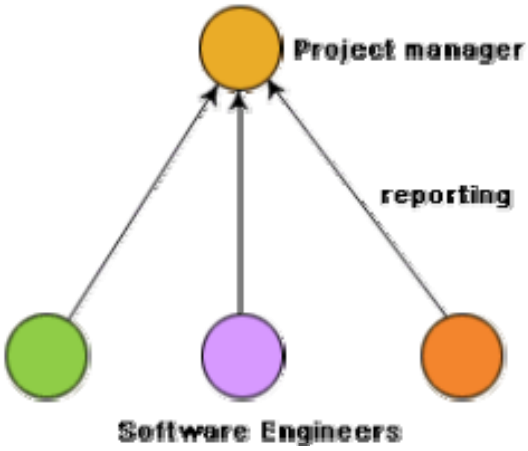


Figure: Chief programmer team structure

- The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking.
- The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

- The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can work out a satisfactory design and ask the programmers to code different modules of his design solution.
- For simple and well-understood problems, an organization must be selective in adopting the chief programmer structure.
- The chief programmer team structure should not be used unless the importance of early project completion outweighs other factors such as team morale, personal developments, life-cycle cost etc.

2. Democratic Team

- The democratic team structure, as the name implies, does not enforce any formal team hierarchy. Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

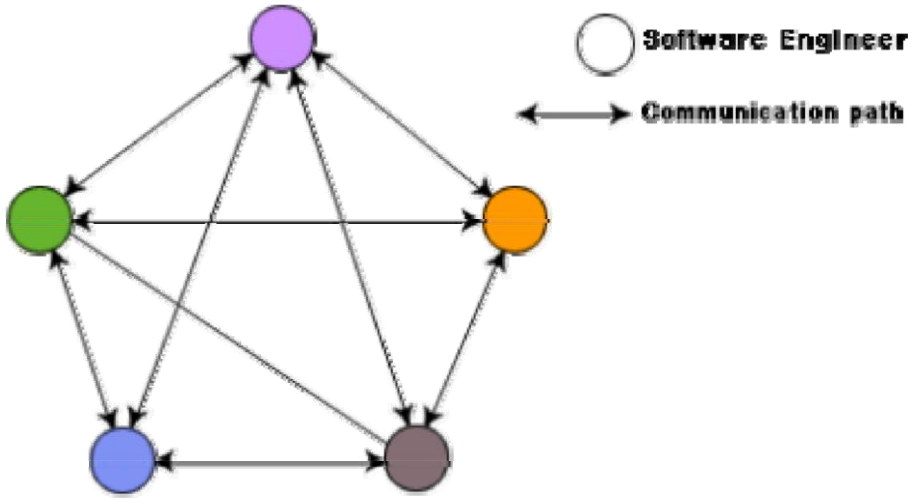


Figure: Democratic team structure

- The democratic organization leads to higher morale and job satisfaction. Consequently, it suffers from less man-power turnover.
- Democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team.
- A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic.
- The democratic team organization encourages egoless programming as programmers can share and review one another's work.

3. Mixed Control Team Organization

- The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization.
- The mixed control team organization is suitable for large team sizes. The democratic arrangement at the senior engineer's level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organization is eminently suited to handle large and complex programs.
- This team structure is extremely popular and is being used in many software development companies.
- The mixed control team organization is shown below. This team organization incorporates both hierarchical reporting and democratic set up. In figure, the democratic connections are shown as dashed lines and the reporting structure is shown using solid arrows.

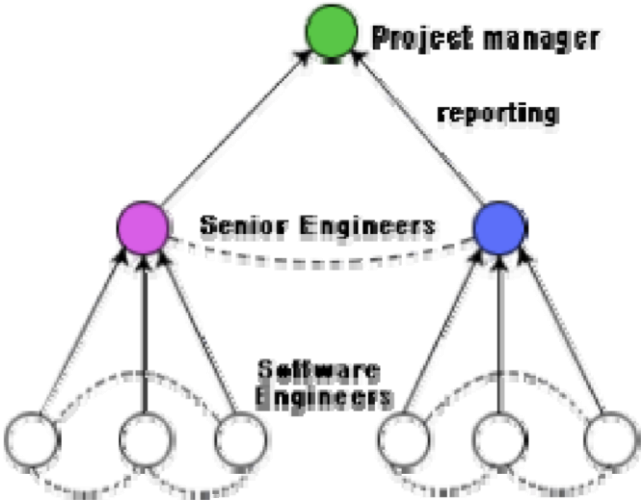


Figure: Mixed team structure

CHARACTERISTICS OF A GOOD SOFTWARE ENGINEER

The attributes that good software engineers should possess are as follows:

- Exposure to systematic techniques, i.e. familiarity with software engineering principles.
- Good technical knowledge of the project areas (Domain knowledge).
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team.
- Discipline, etc.

14. SOFTWARE PROJECT COST ESTIMATION & PROJECT SCHEDULING

COST ESTIMATION

Cost estimation can be defined as the approximate judgement of the costs for a project.

- Cost estimation is usually measured in terms of effort. The most common metric used is person months or years (or man months or years). The effort is the amount of time for one person to work for a certain period of time.
- A cost estimate done at the beginning of a project will help determine which features can be included within the resource constraints of the project (e.g., time). Requirements can be prioritized to ensure that the most important features are included in the product. The risk of a project is reduced when the most important features are included at the beginning because the complexity of a project increases with its size, which means there is more opportunity for mistakes as development progresses. Thus, cost estimation can have a big impact on the life cycle and schedule for a project.
- Cost estimation can also have an important effect on resource allocation. It is prudent for a company to allocate better resources, such as more experienced personnel, to costly projects.

Metrics for software project size estimation

- Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed.
- Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

1. **Lines of Code (LOC)**
   - LOC is the simplest and most widely used metric to estimate project size. The project size is estimated by counting the number of source instructions in the developed program. Lines used for commenting the code and the header lines should be ignored.
   - Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program.

- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.

- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed.

2. **Function point (FP)**

- Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.
- The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data.
- For example, the issue book feature (as shown in below figure) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.

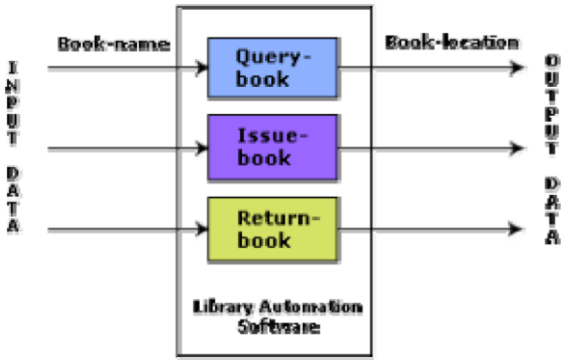


Figure: System function as a map of input data to output data

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance. Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input.

For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the un-adjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, FP=UFP*TCF.

3. **Feature point metric**
   - A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of

developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

- ❖ Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

COCOMO (Constructive Cost Model)

- ➢ COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. COCOMO is used to estimate the total effort required to develop a software project.
- ➢ COCOMO divides the software or projects into three categories.

1. **Organic:** A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.
2. **Semidetached:** A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.
3. **Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.
   - ➢ According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\textbf{Effort} = c1 \times (\text{KLOC})^{c2} \ \textbf{PM}$$

$$\textbf{Tdev} = c3 \times (\text{Effort})^{c4} \ \textbf{Months}$$

Where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- c1,c2,c3,c4 are constants for each category of software products,
- Tdev is the estimated time to develop the software, expressed in months,
- Effort is the total effort required to develop the software product, expressed in person months (PMs).

Software category	c1	c2	c3	c4
Organic	2.4	1.05	2.5	0.38
Semi- Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table: Estimated value of constants for various software categories

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in below figure). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve.

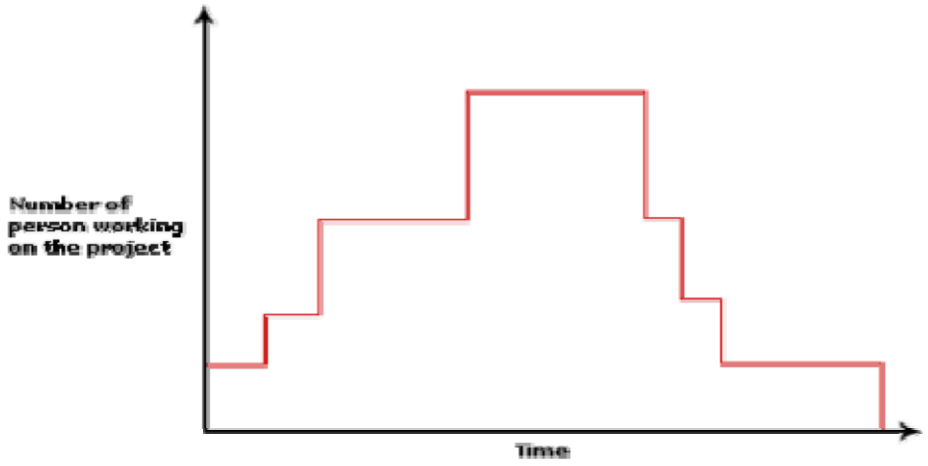


Figure: Person-month curve

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : **Effort = 2.4(*KLOC*)$^{1.05}$ PM**

Semi- detached : **Effort = 3.0(*KLOC*)$^{1.12}$ PM**

Embedded : **Effort = 3.6(*KLOC*)$^{1.20}$ PM**

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : **Tdev = 2.5(*Effort*)$^{0.38}$ Months**

Semi- detached : **Tdev = 2.5(*Effort*)$^{0.35}$ Months**

Embedded : **Tdev = 2.5(*Effort*)$^{0.32}$ Months**

The effort is somewhat super-linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size as shown in the graph.

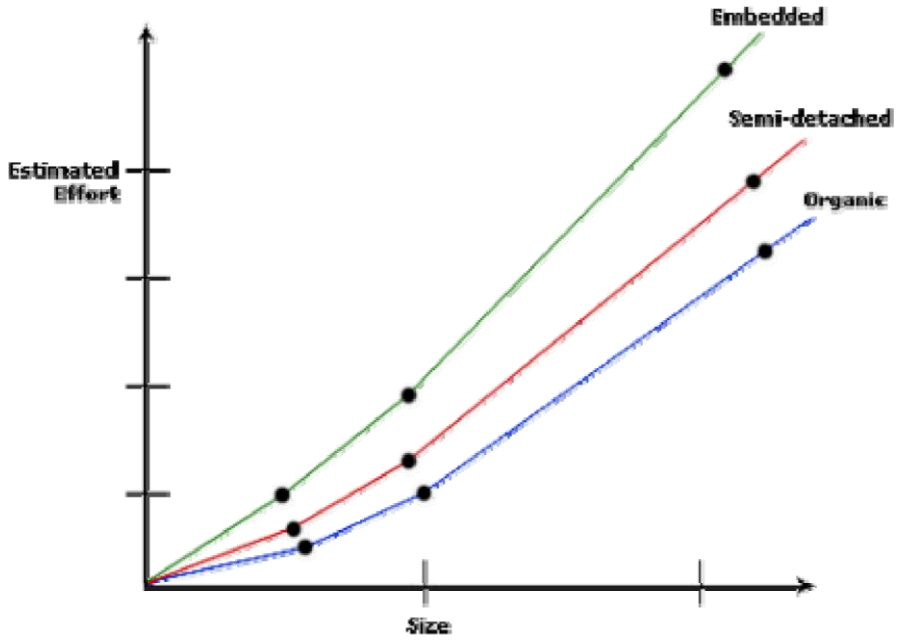


Figure Effort versus product size

The development time is a sub-linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. From the graph below, it can be observed that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

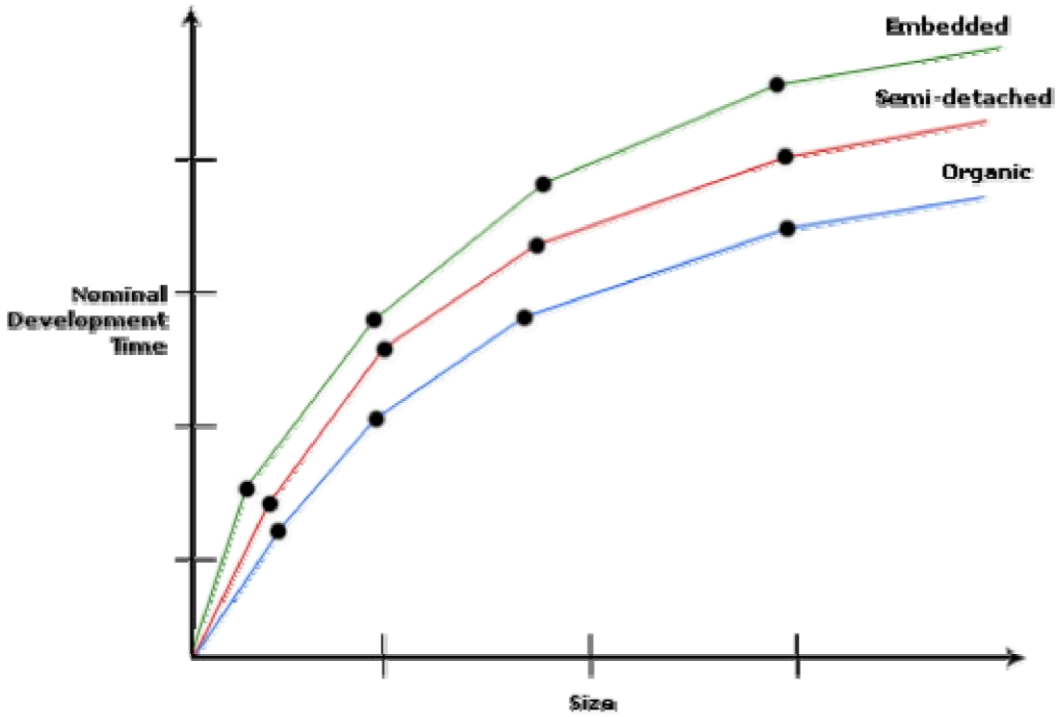


Fig. 11.5: Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

STEPS TO ESTIMATE COST IN BASIC COCOMO

1. Identify the category to which the software belong to.
2. Estimate the Lines of Code(LOC) and convert it into KLOC
3. Estimate the effort using the appropriate equation constants c1,c2.
4. Estimate the time for development using the computed effort value.
5. Then estimate the development cost

Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **$2.4 \times (32)^{1.05} = 91$ PM**

Nominal development time = **$2.5 \times (91)^{0.38} = 14$ months**

Cost required to develop the product = **$14 \times 15{,}000$**

= **Rs. 210,000/-**

INTERMEDIATE COCOMO

- Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

- Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of four "cost drivers" each with a number of subsidiary attributes:-

1. **Product:** The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.
2. **Computer:** Characteristics of the computer that are considered include the execution speed required, storage space required, memory constraints etc.
3. **Personnel:** The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.
4. **Development Environment:** Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

COMPLETE COCOMO

- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These subsystems may have widely different characteristics.
- The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.
- The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

- Database part
- Graphical User Interface (GUI) part
- Communication part

➢ Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

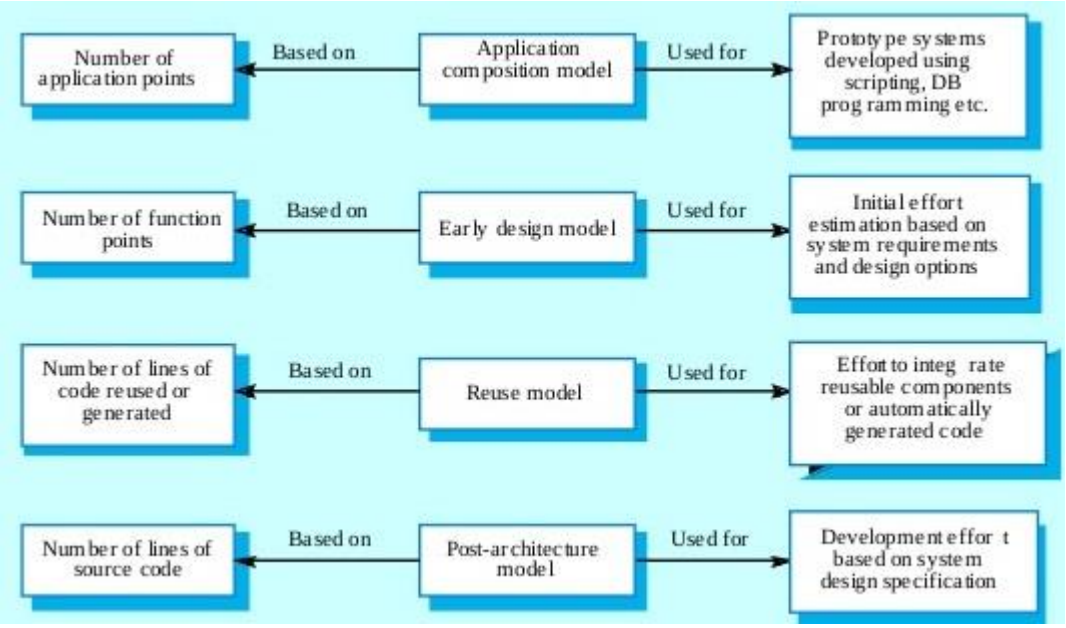
COCOMO – 2

➢ Basic COCOMO is not suitable for projects of larger size as well as projects which uses reuse – approach. COCOMO – 2 deals with the above cases and is most suitable for projects developed in Third Generation Languages (3GL).

➢ COCOMO – 2 is used to estimate project costs at different phases of the software. As the project progresses, though these models can be applied at different stages of the same project.

➢ The various models in COCOMO – 2 are

1. **Application composition:** Used to estimate cost for prototyping.
2. **Early design:** Used to estimate cost at the architectural design stage.
3. **Post –Architecture :** Used to estimate cost during detailed design and coding stage.
4. **Reuse model:** Used to estimate cost, if code reuse is adapted.



➢ **STEPS TO ESTIMATE EFFORT IN APPLICATION COMPOSTION**

1. Estimate the number of screens, reports and 3GL components from an analysis of the SRS document.
2. Determine the complexity level of each screen and report, and rate these as either simple, medium or difficult. The complexity of a screen or report is determined by the number of tables and views it contains.

Table for Screen complexity assignment			
Number of views	**Tables <4**	**Tables<8**	**Tables>=8**
<3	Simple	Simple	Medium
3-7	Simple	Medium	Difficult
>8	Medium	Difficult	Difficult
Table for Report complexity assignment			
Number of sections	**Tables <4**	**Tables<8**	**Tables>=8**
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 or more	Medium	Difficult	Difficult

3. Estimate the complexity and find out the equivalent weight value using the below table. Weights are the amount of effort required to implement an instance of an object at the assigned complexity class.

Table for complexity weights for each class of objects			
Object type	**Simple**	**Medium**	**Difficult**
Screen	1	2	3
Report	2	5	8
3GL components	--	--	10

4. Determine the number of object points (OP). The object point count is the sum of all the assigned complexity values for the object instances together.
5. Estimate the expected percentage of reuse in the system. Then evaluate the number of New object point count (NOP).

NOP= ((Object-Points)*(100-% of reuse)) / 100

6. Determine the productivity rate, PROD = NOP / person – month based on CASE maturity value.
7. Finally Effort is computed as E = NOP / PROD.

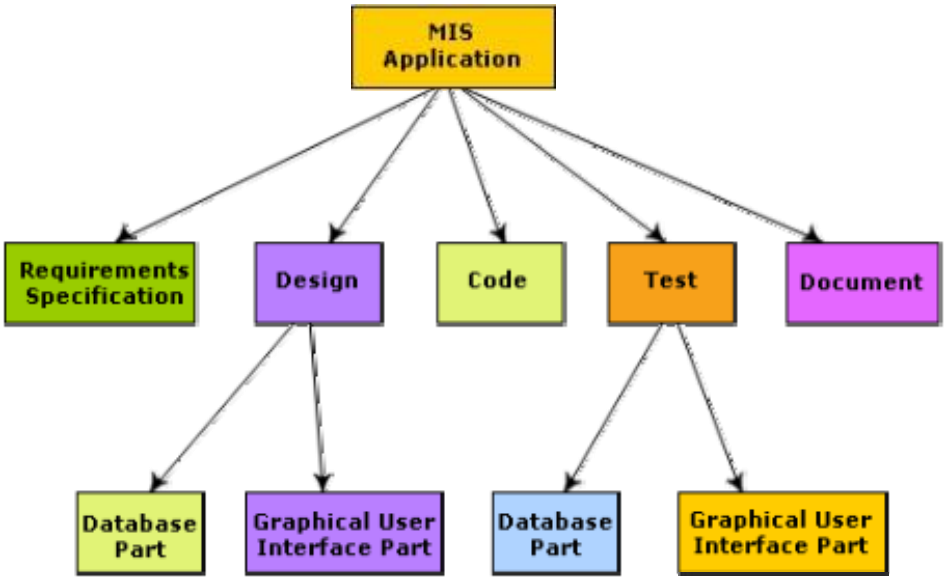
SOFTWARE PROJECT SCHEDULING

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

WORK BREAK DOWN STRUCTURE

- Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem.
- The root of the tree is labelled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. The following figure represents the WBS of an MIS (Management Information System) software.



GANTT CHART

- The simplest project management tool used to represent the timeline of activities is the Gantt chart. Henry L. Gantt lent his name to a simple and very useful graphical representation of a project development schedule. The Gantt chart shows almost all of the information contained in the schedule activity list, but in a much more digestible way. The schedule information is more easily grasped and understood, and the activities can be easily compared.
- The Gantt chart enables us to see at any given time, which activities should be occurring in the project. A Gantt chart has horizontal bars plotted on a chart to represent a schedule.
- In a Gantt chart, Time is plotted on the horizontal axis and activities on the vertical axis. An activity is represented by a horizontal bar on the Gantt chart. The position of a horizontal bar shows the start and end time of an activity and the length of the bar show its duration. Gantt chart can be used to analyse the progress of the project.

ID	Task Name	Expected Start	Expected Finish	Duration
1	Requirement Gathering & analysis	02-11-2015	12-11-2015	1w 4d
2	Architectural and Detailed design	13-11-2015	30-11-2015	2w 2d
3	Database & GUI Design	18-11-2015	23-11-2015	4d
4	Module 1 implementation	01-12-2015	15-12-2015	2w 1d
5	Module 2 implementation	02-12-2015	30-12-2015	4w 1d
6	Unit Testing	02-12-2015	30-12-2015	4w 1d
7	Integration and System Testing	31-12-2015	18-01-2016	2w 3d
8	Documentation	02-11-2015	25-01-2016	12w 1d

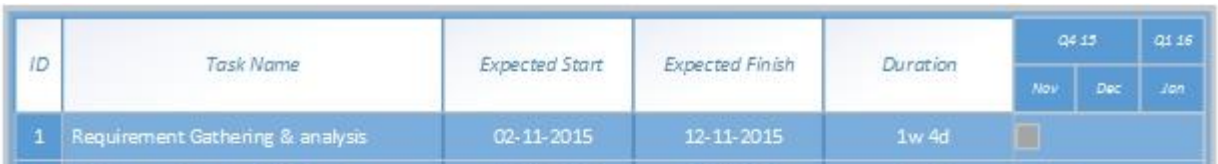


Figure: GANTT chart

PERT CHART

- ✓ PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies.
- ✓ PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made.
- ✓ Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex.
- ✓ A critical path in a PERT chart is shown by using thicker arrows.
- ✓ Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.

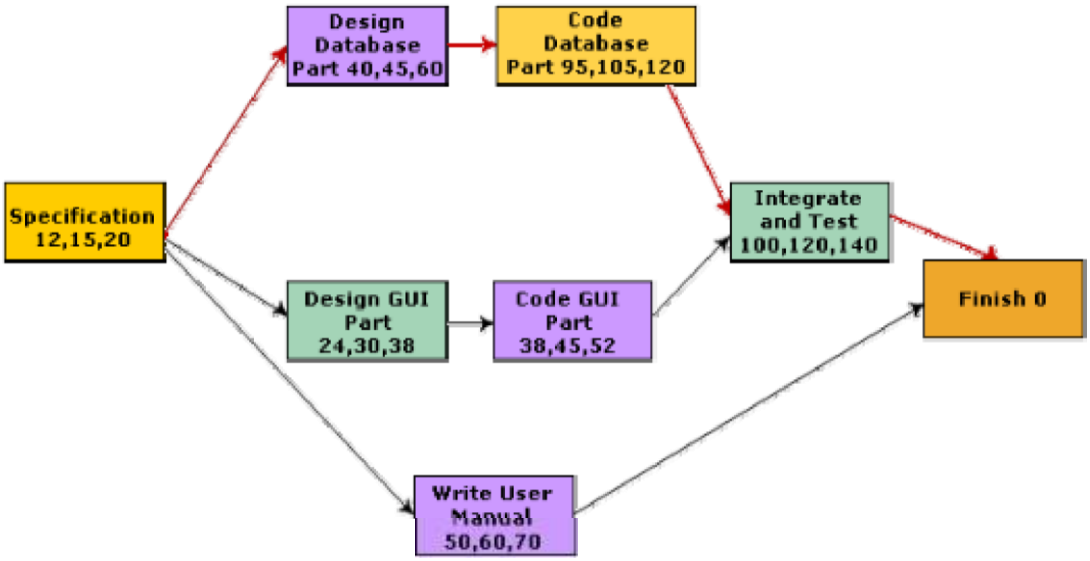


Figure: PERT chart representation of the MIS problem

15. CASE

CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.

Benefits of CASE tools

- Reduces the software development time and cost by automating many repetitive manual tasks.
- Helps to create good quality documentation and thus provides a better quality product.
- Helps to create more maintainable software systems.
- Reduces the burden of software engineer.
- Provides more structured and ordered development methodology.

Characteristics of a successful CASE tool

- It should support standard software development methodology and modelling techniques.
- It must provide an integrated environment for software development.
- It must be flexible, so that user can make necessary changes.
- It should support reverse engineering process.
- It must support integration with automated testing tools.
- It must provide online help.

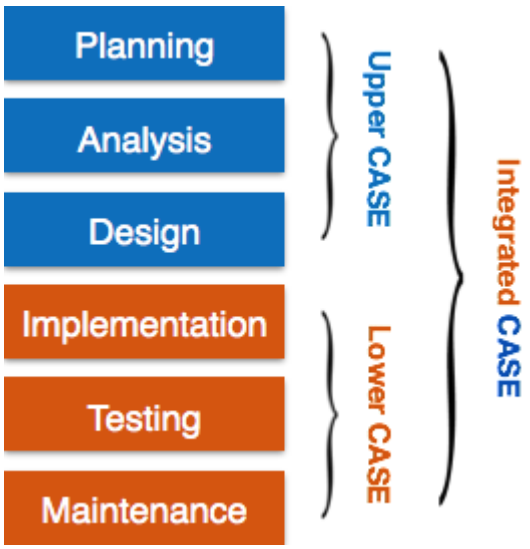
CASE CLASSIFICATIONS

- **Upper Case Tools** – Tools that mainly concentrate on high level activities of SDLC. Upper CASE tools are used in planning, analysis and design stages of SDLC.

- **Lower Case Tools** – tools mainly focuses on the implementation of the system. Lower CASE tools are used in implementation, testing and maintenance.

- **Integrated Case Tools** - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.



CASE ENVIRONMENT

- A schematic representation of a CASE environment is shown below

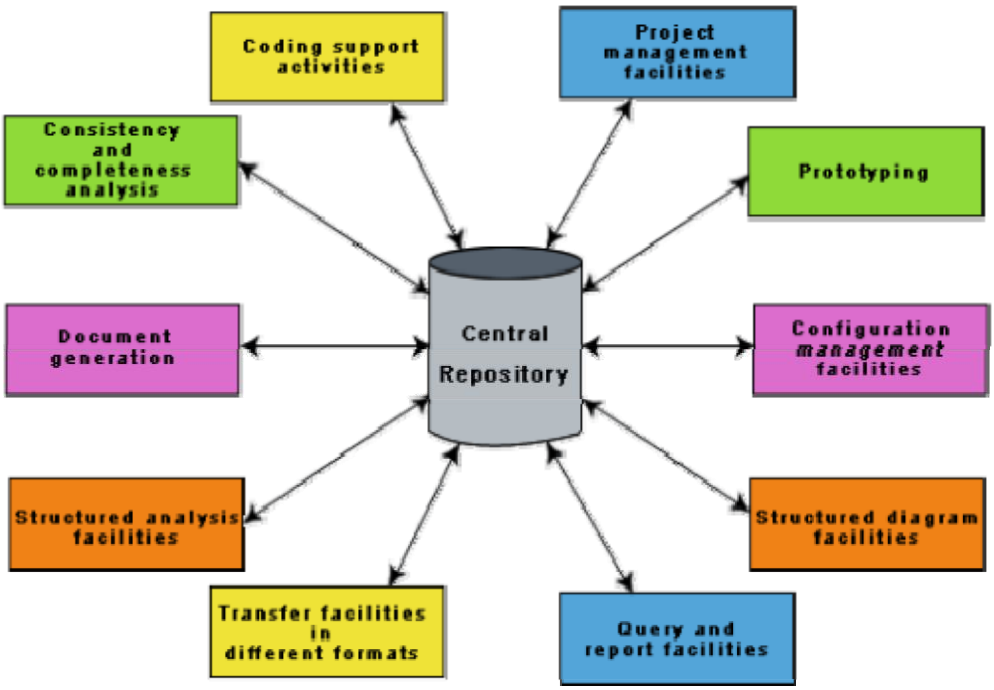


Figure A CASE Environment

- Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts.
- This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment

share common information among themselves. Thus a CASE environment facilities the automation of the step-by-step methodologies for software development.

- **Central Repository** - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management is stored. Central repository also serves as data dictionary.

CASE TOOLS

- CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system.
- There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools are to name a few.
- Use of CASE tools accelerates the development of project to produce desired result and helps to uncover flaws before moving ahead with next stage in software development.
- CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage:
- Various CASE tools are listed below:

1. **Diagram tools**
   These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form. For example, Flow Chart Maker tool for creating state-of-the-art flowcharts.
2. **Process Modelling Tools**

   Process modelling is method to create software process model, which is used to develop the software. Process modelling tools help the managers to choose a process model or modify it as per the requirement of software product. For example, EPF Composer

3. **Project Management Tools**

   These tools are used for project planning, cost and effort estimation, project scheduling and resource planning. Managers have to strictly comply project execution with every mentioned step in software project management. Project management tools help in storing and sharing project information in real-time throughout the organization. For example, Creative Pro Office, Trac Project, Basecamp.

4. **Documentation Tools**

   Documentation in a software project starts prior to the software process, goes throughout all phases of SDLC and after the completion of the project.

   n tools generate documents for technical users and end users. Technical users are mostly in-house professionals of the development team who refer to system manual, reference manual, training manual,

installation manuals etc. The end user documents describe the functioning and how-to of the system such as user manual. For example, Doxygen, DrExplain, Adobe RoboHelp for documentation.

5. **Analysis Tools**

   These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions. For example, Accept 360, Accompa, Case-Complete for requirement analysis, Visible Analyst for total analysis.

6. **Design Tools**

These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques. These tools provides detailing of each module and interconnections among modules. For example, Animated Software Design

7. **Configuration Management Tools**

   An instance of software is released under one version. Configuration Management tools deal with –

   Version and revision management

   Baseline configuration management

   Change control management

   CASE tools help in this by automatic tracking, version management and release management. For example, Fossil, Git, Accu REV.

8. **Change Control Tools**

These tools are considered as a part of configuration management tools. They deal with changes made to the software after its baseline is fixed or when the software is first released. CASE tools automate change tracking, file management, code management and more. It also helps in enforcing change policy of the organization.

9. **Programming Tools**

These tools consist of programming environments like IDE (Integrated Development Environment), in-built modules library and simulation tools. These tools provide comprehensive aid in building software product and include features for simulation and testing. For example, Cscope to search code in C, Eclipse.

10. **Prototyping Tools**

Software prototype is simulated version of the intended software product. Prototype provides initial look and feel of the product and simulates few aspect of actual product.

Prototyping CASE tools essentially come with graphical libraries. They can create hardware independent user interfaces and design. These tools help us to build rapid prototypes based on existing information. In addition, they provide simulation of software prototype. For example, Serena prototype composer, Mockup Builder.

11. **Web Development Tools**

These tools assist in designing web pages with all allied elements like forms, text, script, graphic and so on. Web tools also provide live preview of what is being developed and how will it look after completion. For example, Fontello, Adobe Edge Inspect, Foundation 3, Brackets.

12. Quality Assurance Tools

Quality assurance in a software organization is monitoring the engineering process and methods adopted to develop the software product in order to ensure conformance of quality as per organization standards. QA tools consist of configuration and change control tools and software testing tools. For example, SoapTest, AppsWatch, JMeter.

13. Maintenance Tools

Software maintenance includes modifications in the software product after it is delivered. Automatic logging and error reporting techniques, automatic error ticket generation and root cause Analysis are few CASE tools, which help software organization in maintenance phase of SDLC. For example, Bugzilla for defect tracking, HP Quality Center.

Architecture of a CASE environment

- The architecture of a typical modern CASE environment is shown below. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository. Characteristics of a tool set have been discussed earlier.

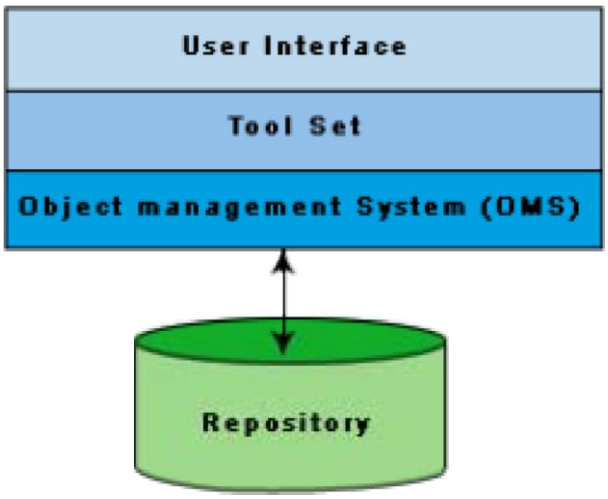


Figure: Architecture of a Modern CASE Environment

User Interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

Object Management System (OMS) and Repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities

such into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus the object management system takes care of appropriately mapping into the underlying storage management system.

WORKBENCHES

A coherent set of tools that is designed to support related software process activities such as analysis, design or testing

- Workbenches integrate several CASE tools into one application to support specific software-process activities. Hence they achieve

- ➢ A homogeneous and consistent interface (presentation integration).
- ➢ Easy invocation of tools and tool chains (control integration).
- ➢ Access to a common data set managed in a centralized way (data integration).

- CASE workbenches can be further classified into following 8 classes:

1. Business planning and modelling
2. Analysis and design
3. User-interface development
4. Programming
5. Verification and validation (Testing Workbench)
6. Maintenance and reverse engineering
7. Configuration management
8. Project management

ANALYSIS & DESIGN WORKBENCH

- Analysis and design workbenches support system modelling during both requirements engineering and system design.
- These workbenches may support a specific design method or may provide support for a creating several different types of system models.
- Analysis & Design workbench performs various analysis required during the development phase. It will automatically generate the various software design forms like DFDs, UML diagrams, Algorithms etc.
- It will automatically generate suitable documents associated with analysis and design phases of the software.
- The various components of Analysis & Design workbench is shown below:

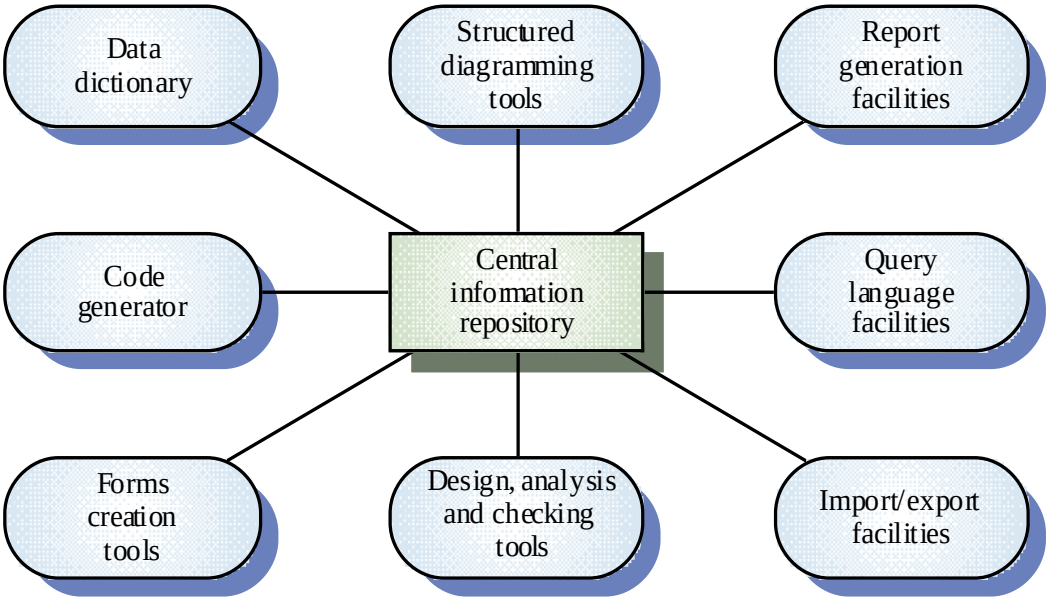
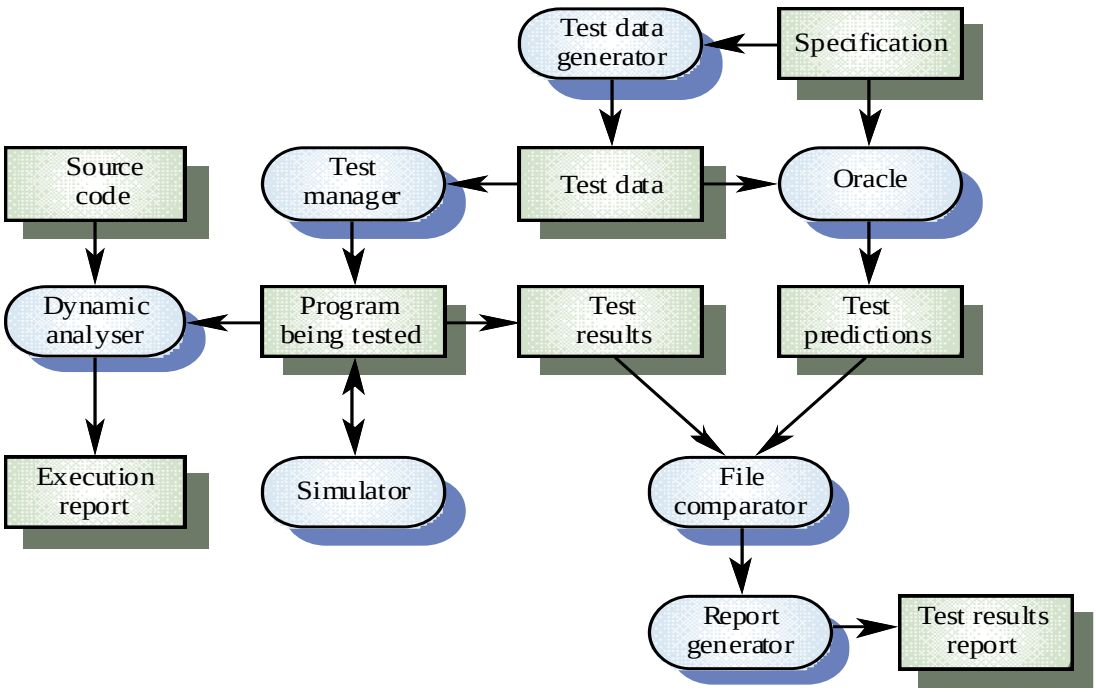


Figure: Components of Analysis & Design Workbench

TESTING WORKBENCH

- Testing is an expensive process phase.
- Testing workbenches provide a range of tools to reduce the time required and total testing costs
- Most testing workbenches are open systems because testing needs are organization-specific
- Difficult to integrate testing with closed design and analysis workbenches.
- Scripts may be developed for user interface simulators and patterns for test data generators
- Test outputs may have to be prepared manually for comparison
- The various components of testing workbenches are shown below:



REFERENCES

Text books

1. Fundamentals of Software Engineering – Rajib Mall
2. Software Engineering Principles & Practices – Rohit khurana
3. Software Engineering A Practioner's approach – Roger S Pressman
4. Software Engineering –Sommerville
5. An integrated Approach to Software Engineering – Pankaj Jalote

Websites

1. **http://istqbexamcertification.com/**