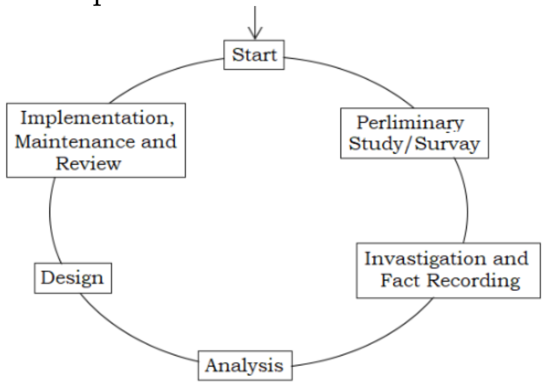
1. Explain SDLC with different phases.

SDLC (system development life cycle) consist of setup development activities that have been prescribed order.

The SDLC is classically thought of as the set of activities that analyst, designers and users carry out to develop and implement an information system.

The following figure shows different phases of SDLC –


Start
Perliminary Study/Survay
Invastigation and Fact Recording
Analysis
Design
Implementation, Maintenance and Review

1) Preliminary Study or Survey:

This is the first step of SDLC. This phase is basically considered about determining whether or not new system should be developed.

During this phase, problems with current system are identified and also the benefits and cost of an alternative system are identified. If the benefit seems to outwait the cost, the approval is granted for new product development.

a) Feasibility Study:

The basic purpose of feasibility study or survey is to determine whether the whole processes of system analysis leading to computerized would be worth the effort for the organization. For instance, if an organization is attempting to integrate a number of different computerized system, the feasibility might reveal then fact that the integration would not yield desired results and it is batter to design the system from scratch. The feasibility study, results in the preparation of a report called the *feasibility study / survey report.*

It contains the following details—

i) A proposed solution on the problems including alternate solutions considered.
ii) Rough estimate on the cost/benefit analysis if the solution is implemented.
iii) A approximate effort/cost estimate for completion of the project.

***) Cost benefit analysis:** To carry out an economic feasibility study it is necessary to actuate money value against any purchase for activities needed to implement the project. It is also necessary to place the money value against a benefit that will acquire from a new system created by the project. Such calculations are described as cost benefit analysis.

Cost benefit analysis usually includes two steps—

i) Producing the estimate of cost and benefit.
ii) Determine whether the project is worthwhile once these cost are ascertained.

2) Investigation and Fact Recording:

After solution is approved, the work of the system development team begins with a careful easement of the needs that the new information system is expected to fulfill.

Common fact recording methods are—

i) Interviewing: Interviews by far is the most common and most satisfactory way of obtaining the information, particularly to obtain information about objectives, constrains, allocation of benefit and problems and failures in the existing system.

ii) Questionnaire: It might not as effective as direct interviewing. However, in number of situation it may not be possible to interview all the concerned persons on account of their scattered locations or their number beings too large.

Questionnaire designing is critical. Questions should be easy to understand, unbiased, non-threatening and specific.

iii) Onsite observation: It is processes of recording and voting people, objectives and occurrence to obtain information. The major objective of onsite observation is to get as close as possible to the real system being studied.

iv) Sampling: Where the volume of transaction is large it may not be possible to study the entire set of documents to arrive at certain conclusion. However, one should be careful in selecting the sample of the system in such a way that it would be helpful to develop the full product.

3) System Analysis:

System analysis follows the feasibility study. This the period when the development team collects information from user. Management and data processing personal regarding the proposed system and the environment in which it will operate, then analyses this information in order to specify a document requirement specification. These specification states—

i) Goals and objectives of the proposed system.
ii) Fundamental action that must take place in the software.
iii) Outputs to be produced.
iv) Inputs to be provided.
v) Processes to be executed.
vi) Interfaces to be provided.
vii) Performance requirement to be made.
viii) Organizational and other constrains to be made.

4) System Design:

This is the most creative and challenging phase of SDLC. This phase is concerned with the design of final system. In this phase, firstly the output is designed keeping in mind how the output is to be produced and in what format. Secondly, input data and files are designed to meet the requirements of the proposed output. The processing design involves program construction and testing.

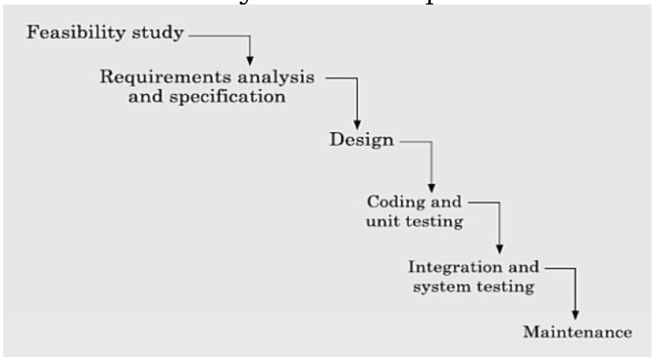
5) Implementation, Maintenance and Review:

These are post design phase; all these are less creative than system design. Implementation phase is mainly concerned with user testing, site preparation and file conversion. It also involves final testing of the system

Once the system is implemented; evaluation and maintenance begins.

2. Explain Classical Waterfall Model with its different phases.

The classical waterfall model is intuitively the most obvious way to develop software. The classical waterfall model divides the life cycle into the phase as shown in the following figure—


Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

Observer that diagrammatic representation of this model resembles a cascade of waterfall. This resembles possibly justifies the name of this model.

The classical waterfall model brakes down the life cycle into an intuitive set of phases. The different phases of this model are: feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance. The phases starting from the feasibility study to the integration and system testing phase are known as the development phases. The maintenance phases commence after the completion of the development phases.

1) Feasibility study:

The main focus of the feasibility study stage is to determine whether it would be financially and technically feasible to develop the software. The feasibility study involves analysis of the problem and collection of all relevant information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analyzed to perform the following—

i) An abstract problem definition.
ii) Formulation of the different possible strategies for solving the problem.
iii) Evaluation of different solution strategies.

2) Requirements analysis and specification:

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities—

i) Requirement gathering and analysis: This activity consist of first gathering the requirements and then analysis of the gathered requirements. The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the customer's requirements. Once the requirements are gathered, the analysis is taken up. The goal of the requirements analysis activity is to check out the incompleteness and inconsistencies in these gathered requirements.

ii) Requirements specification: After the requirement gathering and analysis activity, the identified requirements are documented. This is called a *software requirements specification* (SRS) document. The SRS document should be easy to understand. The most important contains of this document are the functional and non-functional requirements and the ultimate goal of the implementations.

3) Design:

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are being used at presents. These approaches are—

i) Procedural/Traditional design approach.
ii) Object-oriented design approach.

4) Coding and Unit Testing:

The purpose of the coding and unit testing phase is to translate a software design into source code. The coding phase is also sometimes called the *implementation phase*. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually unit tested.

After the coding is completed, each module is unit tested. Unit testing involves testing each module in isolation from other module, then debugging and documenting it.

5) Integration and System:

Integration of different modules is undertaken soon after they have been coded and unit tested. During the integration and system testing phase, the different modules are integrated in a planned manner. Integration of various modules are normally carried out incrementally over a number of steps. The goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

System testing usually consists of three different kinds of testing activities:

i) alpha-testing: Performed by the development team.
ii) beta-testing: Performed by a friendly set of customer.
iii) Acceptance testing: Performed by the normal users.

6) Maintenance:

Maintenance of a typical software product requires much more effort than the effort necessary to develop the product itself. Maintenance involves performing any one or more of the following three kinds of activities—

i) Corrective maintenance.
ii) Perfective maintenance.
iii) Adaptive maintenance.

Advantages:

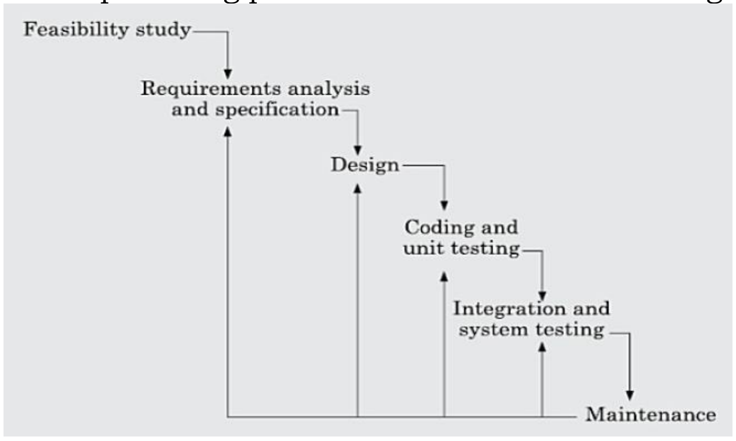
1. The way structuring different activities in a software project appears simple and logical.
2. Every stage produces a concrete deliverable such as requirement documents code etc.
3. It facilitated software contracting. Requirement specification help in establishing contract. The deliverable cab be tested against specification.
4. It provides a simple measurement of the development status. Progress from one phase to next phase indicates achievement of an important milestone.

Disadvantages:

1. It is difficult to accurately and completely specify requirements prior to any implementation.
2. Once the requirement specifications are established in the analysis stage it is difficult to make any changes in subsequent stages. Hence, the users are locked in to features that may not reflect their needs.
3. A working version of the software is not available until the acceptance testing stage.
4. The phases of development do not necessarily reflect progress toward completion. Entering the test stage may mean anything from 20% to 80% completion.

3. Explain Iterative Waterfall Model.

In a practical software development work, it is not possible to strictly follow the classical waterfall model. Therefore, the classical waterfall model is branded as an idealistic model. In this context we can view the classical waterfall model as making necessary changes to the classical waterfall model, so that it becomes applicable to practical software development projects. Essentially the main changes to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases as shown in the following figure—


Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

The feedback paths allow for correction of the errors committed during a phase, as and when these are detected in a later phase. For example, if during testing a design error is identified then the feedback path allows the design to be reworked and the changes to be reflected in the design documents however, observe that there is no feedback path to the feasibility stage. This mean that the feasibility study errors can’t be corrected.

The iterative waterfall model consists of the following six phases—

Advantages:

5. In iterative waterfall model requirements may be modified because of feedback paths.

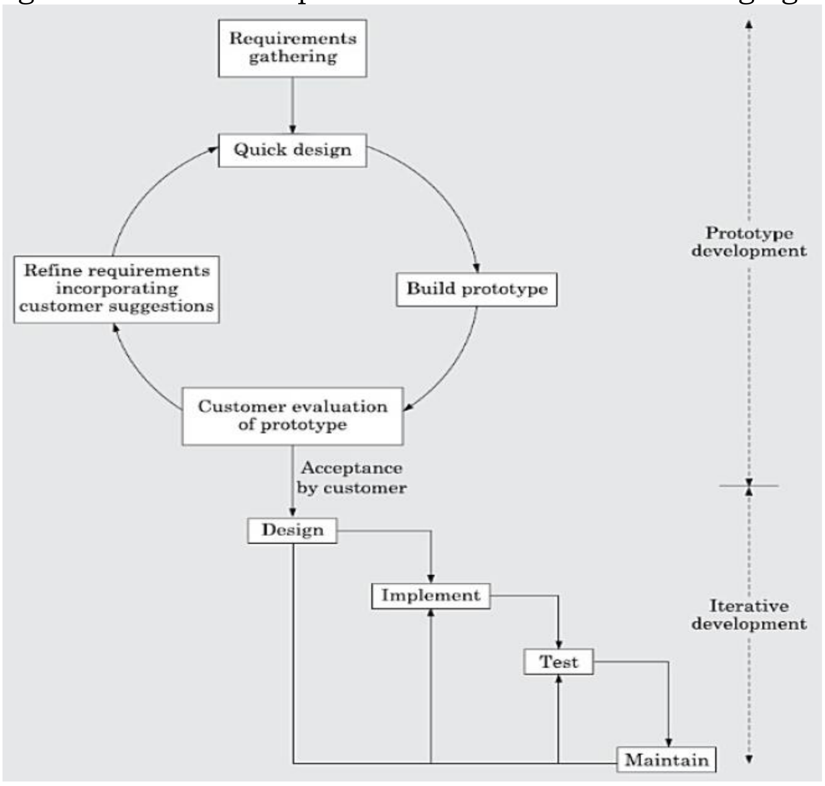
Disadvantages:

1. The iterative waterfall model can't satisfactorily handle the different types of risks that a real life software project may suffer from.
2. To achieve better efficiency and higher productivity most real life projects find it difficult to follow the rigid phases sequence prescribed by the iterative waterfall model. By the team a rigid phases sequence we mean that a phases consist only offer the previous phases is complete in all respects.

4. Explain Prototyping Model.

The 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 of the system. A prototype is usually built using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up rather than by performing the actual computations. A prototype usually turns out to be a very crude version of the actual system, possibly exhibiting limited functional capabilities, low reliability and inefficient performance as compared to the actual software.

The prototyping of software development is shown in the following figure—



As shown in the following figure, the first phase is prototype development control various risk this is followed by an iterative development cycle.

1) **Prototype development:** Prototype development starts with an initial requirements gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for 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.

2) **Iterative development:** Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase becomes redundant since the

working prototype that has been approved by the customer serves as an animated requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system. Therefore, even though the construction o f a throwaway prototype might involve incurring additional cost, for systems with unclear customer requirements and for systems with unresolved technical issues, the overall development cost usually turns out to be lower compared to an equivalent system developed using the iterative waterfall model. By constructing the prototype and submitting it for user evaluation, many
customer requirements get properly defined and technical issues get resolved by experimenting with the prototype. This minimizes later change requests from the customer and the associated redesign costs.

Advantages:

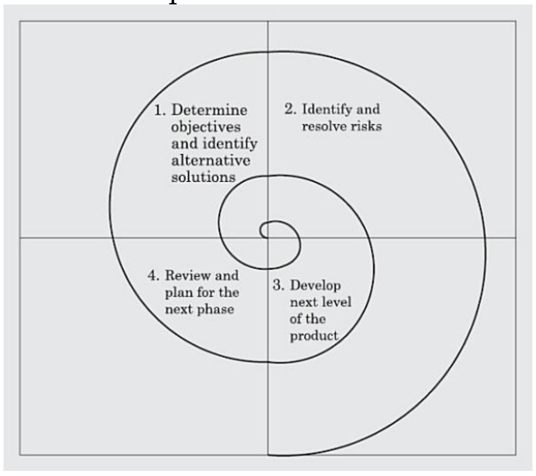
1. Instead of concentrating on documentation, more effort id placed in creating the actual software.
2. The software is created by using lots of feedback from users. Every prototype is created to get the view and opinions about the software from users.
3. Since the user are involved in software development, there is a great chance of the resulting software being accepted to theme.
4. If something unfavorable, it can be changed.
5. Overdesigning could be also avoided using this model. “Overdesign” happens when the software has so many features that the focus on core requirements are lost. Prototyping facilitated on giving only what users wants.

Disadvantages:

1. Often users feel that a few changes will sufficient for their needs, they do not realize that the prototype is meant only for demonstration purpose and it is not real software. A lot of other work such as algorithm design, coding, debugging and quality related activities are done for developing the real software.
2. The developer may lose focus on the real purpose of the prototype and compromise on the quality of the product.
3. A prototype has no legal standing in the event if any despite. Hence the prototype as the software specification is used only as a tool software development and not as a part of the contract.

5. Explain Spiral Model briefly.

The spiral model of software development is shown in the following figure—


1. Determine objectives and identify alternative solutions
2. Identify and resolve risks
3. Develop next level of the product
4. Review and plan for the next phase

The diagrammatic representation of this model appears like a spiral with many loops. The exact no of loops of the spiral is not fixed and can vary from project to project. The no of loops shown in the above figure is just an example. Each loop of the spiral is called a phase of the software process. It can be seen that this model is much more flexible compared to other models, since the exact no of phases through which the product is developed is not fixed.

A risk is essentially any adverse circumstance that might hamper the successful completion of a software project. As an example, consider a project for which a risk can be that data access from a remote database might be too slow to be acceptable by the customer. This risk can be resolved by building a prototype of the data access subsystem and experimenting with the exact access rate. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of software development.

Each phase in this model is split into four sectors (or quadrants) as shown in the above figure. In the first quadrant, a few features of the software are identified to be taken up for immediate development based on how crucial it is to the overall software development. Implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analyzed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far (i.e. the developed version of the software) with the customer and planning the next iteration of the spiral.

To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified. With each iteration around the spiral (being at the center and moving outwards) progressively more complicated version of the software get built. The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase. In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses. Therefore, in this model, the project manager plays the crucial role of tuning the model to specific projects.

Advantages:

The spiral model uses a risk management approach to software development. Some advantages of the spiral models are—

i) It differs elaboration of low risk software elements.
ii) It incorporates prototyping as a risk reduction strategy.
iii) It gives an early focus to reusable software.
iv) It accommodates lifecycle evolution, growth and requirements changes.
v) It incorporates software quality objective onto product.
vi) It focuses on early error detecting and design flaws.

Disadvantages:

i) It does not work well with contract work.
ii) It relays on the ability of software engineers to identify source of risk.
iii) The spiral model usually appears as a complex model to follows, since it is risk driven and more complicated.

6. Explain various type of software metrics for the estimation software size.

Accurate estimation of the problem size is fundamental satisfactory estimation of other project parameters such as effort, time duration for completing the project and the total cost for developing the software.

The size of a project is not obviously the no of bytes that the source code occupies, it is neither the byte size of the executable code. The project size is measure of the problem complexity in terms of the effort and time required to develop the product. Currently two metrics are popularly used to estimate the size.

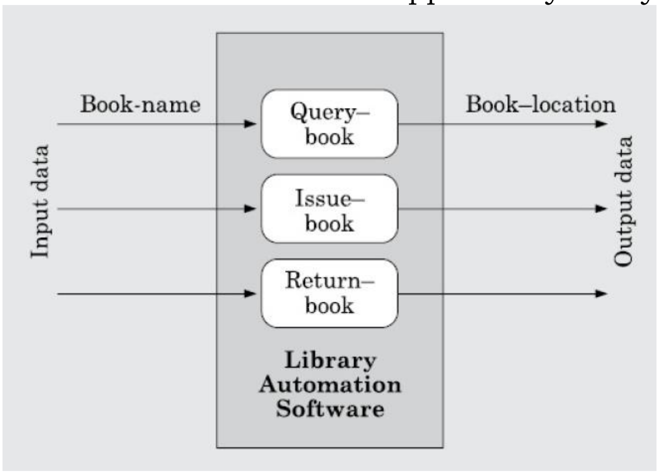
1) Lines of Code (LOC): LOC is possibly the simplest among all metrics available to measure project size. Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.

Determining the LOC count at the end of a project is very simple. However, accurate estimation of LOC count at the beginning of a project is a very difficult task. One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and so on, until the LOC of the leaf-level modules are small enough to be predicted. To be able to predict the LOC count for the various leaf-level modules sufficiently accurately, past experience in developing similar modules is very helpful. By adding the estimates for all leaf level modules together, project managers arrive at the total size estimation. In spite of its conceptual simplicity, LOC metric has several shortcomings when used to measure problem size, discussed in the following—

i) LOC gives a numerical value of problem size that can vary widely with individual coding in different ways.
ii) LOC is a measure of coding activity alone. A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e. specification, 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.
iii) LOC measure correlates poorly with the quality and efficiency of the code.
iv) LOC metric penalizes use of higher-level programming languages and code reuse.
v) LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities.
vi) It is very difficult to accurately estimate LOC of the final program from problem specification.

2) Function Point (FP) Metric: This metric overcomes many of the shortcomings of the LOC metric. Function point metric has several advantages over LOC metric. One of the important advantages of the function point metric over the LOC metric is that it can be used to easily estimate the size of a software product directly from the problem specification.

The conceptual idea behind the FPM is that the size of a software product is directly dependent on the no of different functions on features it supports. A software product supporting many features would certainly be of large size then a product with less no of features. Each function when invoked reads some input data and transform it to the corresponding output data. For example the query book features (Shown in the following figure) of a Library Automation Software takes the name of the book as input and display its location and the no of copies available. Similarly, the issue book and return book features product their outputs based on the corresponding input data. Thus a computation of the no of input and output data values to a system gives some indication of the no of functions supported by the system.



Here in addition to the number of basic functions that a software performs, size also depends on the number of files and the number of interfaces. Here, interfaces refer to the different mechanisms for data transfer with external systems. Besides using the no of input and output data values, FPM computes the size of a software product using 3 other characteristics of the product shown in the following expression.

Function point is computed in three steps—

Step 1: Compute the unadjusted function point (UFP) using a heuristic expression.

Step 2: Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.

Step 3: Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

UFP = (No. of inputs)*4 + (No. of outputs)*5 + (No. of inquiries)*4 + (No. of files)*10 + (No. of interfaces)*10

The expression shows the computation of UFP as the weighted sum of these five problem characterizes. The weights associated with the five characterizes ware validated through the data gathered from many projects. A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield. Finally FP is given as the product of UFP and TCF, that is

FP = UFP x TCF

7. Explain COCOMO Briefly.

COCOMO (COnstructive COst estimation Model) was proposed by Boehm in 1981. Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity—

i) **Organic:** We can classify a development project to be 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.

ii) **Semidetached:** A development project can be classify to be of semidetached type, if the development team 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.

iii) **Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist.

1) **Basic COCOMO:** The basic COCOMO model is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms—

$$\text{Effort} = a_1 \times (\text{KLOC})a_2 \text{ PM}$$
$$T_{dev} = b_1 \times (\text{Effort})b_2 \text{ months}$$

Where,

a) KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
b) a_1, a_2, b_1, b_2 are constants for each category of software product.
c) T_{dev} is the estimated time to develop the software, expressed in months.
d) Effort is the total effort required to develop the software product, expressed in person-months (PMs).

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be n LOC. The values of a1, a_2, b_1, b_2 for different categories of products as given by Boehm.

i) **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 $(\text{KLOC})^{1.05}$ PM
Semi-detached: Effort = 3.0 $(\text{KLOC})^{1.12}$ PM
Embedded: Effort = 3.6$(\text{KLOC})^{1.2}$ PM

ii) **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: T_{dev} = 2.5 $(\text{Effort})^{0.38}$ Months
Semi-detached: T_{dev} = 2.5 $(\text{Effort})^{0.35}$ Months
Embedded: T_{dev} = 2.5 $(\text{Effort})^{0.32}$ Months

2) **Intermediate COCOMO:** The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort as well as the time required to develop the product. For example, the effort to develop a product would vary depending upon the sophistication of the development environment. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimates obtained using the basic

COCOMO expressions. The intermediate COCOMO model uses a set of 15 cost drivers (multipliers) that are determined based on various attributes of software development. These cost drivers are multiplied with the initial cost and effort estimates (obtained from the basic COCOMO) to appropriately scale those up or down. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, the initial estimates are scaled upward. Boehm requires the project manager to rate 15 different parameters for a particular project on a scale of one to three. For each such grading of a project parameter, he has suggested appropriate cost drivers (or multipliers) to refine the initial estimates. In general, the cost drivers identified by Boehm can be classified as being attributes of the following items:

i) **Product:** The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

ii) **Computer:** Characteristics of the computer that are considered include the execution speed required, storage space required, etc.

iii) **Personnel:** The attributes of development personnel that are considered include the experience level of personnel, their programming capability, analysis capability, etc.

iv) **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.

3) **Complete COCOMO:** A major shortcoming of both the basic and the intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up of several smaller sub-systems. These sub-systems often have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some even embedded. Not only may the inherent development complexity of the subsystems be different, but for some subsystem the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. 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 sub-systems.

In other words, the cost to develop each sub-system is estimated separately, and the complete system cost is determined as the subsystem costs. This approach reduces the margin of error in the final estimate.

consider the following development project 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-component:

i. Database Part.
ii. Graphical User Interface (GUI) Part.
iii. 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.

8. What is SRS? Explain different type categories of users of SRS.

After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organize the requirements in the form of an SRS document. The SRS document usually contains all the user requirements in a structured though an informal form.

Different people need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

1) **Users, customers, and marketing personnel:** These persons need to refer to the SRS document to ensure that the system as described in the document will meet their needs. Remember that the customer may not be the user of the software, but may be some one employed or designated by the user. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.

2) **Software developers:** The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.

3) <u>Test engineers:</u> The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate it's working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.

4) <u>User documentation writers:</u> The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.

5) <u>Project managers:</u> The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.

6) <u>Maintenance engineers:</u> The SRS document helps the maintenance engineers to understand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and code. Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.

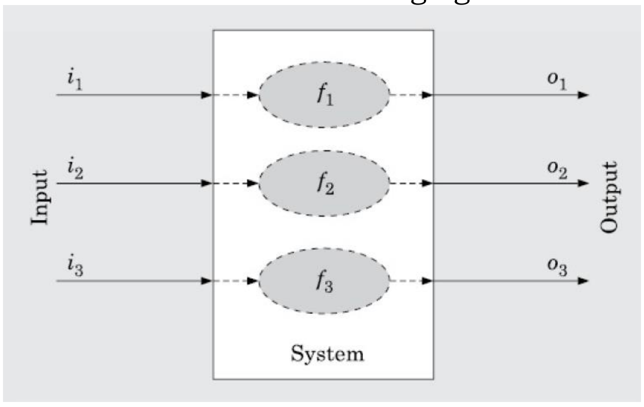
9. Explain The Characteristics of a Good SRS Document.

The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. Some of the identified desirable qualities of an SRS document are the following:

1) <u>Concise:</u> The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.

2) <u>Well Structured/Modifiable:</u> Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. Also, an SRS document is often modified after the project completes to accommodate future enhancements and evolution. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured. A well-structured document is easy to understand and modify. Thus it is important to make the document well structured.

3) <u>Black-Box View:</u> It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behavior of the system and not discuss the implementation issues. This view with which a requirements specification is written, has been shown in the following figure.



In the above figure, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all. The SRS document should describe the system to be developed as a black box, and should specify only the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of the software being developed.

4) <u>Traceable:</u> It should be possible to trace a specific requirement to the design elements that implement it and vice versa. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and vice versa.

Traceability is also important to verify the results of a phase with respect to the previous phase and to analyze the impact of changing a requirement on the design elements and the code.

5) Identification of response to undesired events: The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

6) Verifiable: All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation. A requirement such as "the system should be user friendly" is not verifiable. On the other hand, the requirement—"When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out" is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

10. What are different aspects of SRS Document.

An SRS document should clearly document the following 3 aspects of a system—

1) Functional requirements: The functional requirements capture the functionalities required by the users from the system. It is useful to consider a software as offering a set of functions $\{f_i\}$ to the user. These functions can be considered similar to a mathematical function $f : I \rightarrow O$, meaning that a function transforms an element (i_i) in the input domain (I) to a value (o_i) in the output (O). This functional view of a system is shown schematically in the following figure—

FIG

Each function f_i of the system can be considered as reading certain data ii, and then transforming a set of input data (i_i) to the corresponding set of output data (o_i). The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set.

2) Non-functional requirements: The non-functional requirements are non-negotiable obligations that must be supported by the software. The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data). Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.). The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.

3) Goals of implementation: The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed. These are not binding on the developers, and they may take these suggestions into account if possible. For example, the developers may use these suggestions while choosing among different design solutions.

A goal, in contrast to the functional and non-functional requirements, is not checked by the customer for conformance at the time of acceptance testing.

The goals of implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

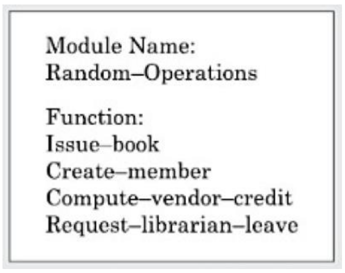
11. What do you mean by Cohesion? Explain the different type of cohesion.

In a good design, all instruction in a module relate to a single function. The extent to which the instruction of a module contributes to performing a single unified task is called cohesion. In other word cohesiveness of a module is the degree to which the different function of the module cooperates to work towards a single objective. The different modules of a design can possess different degrees of freedom.

There are different types of cohesion as follows—

1) Coincidental Cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the

module out of pure coincidence rather than through some thought or design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily. An example of a module with coincidental cohesion has been shown in the following figure—



Module Name:
Random–Operations

Function:
Issue–book
Create–member
Compute–vendor–credit
Request–librarian–leave

Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

2) Functional Cohesion: A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the pay-slips of the employees. When a module possess functional cohesion than we should be able to describe what the module does using only one simple sentence. For example, For the given in the above of employee pay-roll, we describe the overall responsibility of the module by saying "It generates pay-slips of the employees".

3) Logical Cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc. As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

4) Temporal Cohesion: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.

5) Procedural Cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data. Consider the activities associated with order processing in a trading house. The functions login(), place_order(), check_order(), printbill(), place_order_on_vendor(), update_inventory(), and logout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.

6) Communicational Cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.

7) Sequential Cohesion: A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence. As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create_order(), check_item_availability(), place_order_on_vendor() are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create_order() creates an order that is processed by the function check-item-availability() (whether the items are available in the required quantities in the inventory) is input to place_order_on_vendor().

12. What is Coupling? Explain different types of Coupling.

For an integrated system, it is necessary to have some exchange of data between modules. This requirement makes the module somewhat interdependent on one another. The extent to which modules are interdependent is called coupling.

Two modules are said to be highly coupled in either of the two situation arrives—

I. If the functions calls between two modules involves passing large chunks of shared data the modules are tightly coupled.
II. If the interaction occurs through some shared data, then also we say that they are highly coupled.

If two modules either don't interact with each other at all or at best interact by passing no data for only a few preemptive data items, they are said to have low coupling.

They we can say the degree of coupling between two modules depends on their interface complexity.

The following few different types of coupling can exist—

1) Data coupling: Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes. This is the best kind coupling.

2) Stamp coupling: In stamp coupling data are passed in the form of data structures or records. Exchange of data structure instead of data elements tends to make the system more complicated. So, stamp coupling is not as good as data coupling.

3) Control coupling: When one module passes a piece of data or signal to control the action of another module, the two are said to be control coupled. The control information tells the recipient module what action it should perform. Control information may also pass from a subordinate to a super ordinate module. Such a practice is referred as to inversion.

4) Common coupling: When two modules refer to the same global data area, they are common coupled. Global data areas are possible in several programming languages, such as, FORTRAN (Common Bock) and COBOL, the data division is global to any paragraph in the procedure division. The level of module interdependence becomes quite high is such cases.

5) Content coupling: Content coupling involves one module directly referring to the inner work of another module. For example, one module may alter data in a second module or change a statement coded into another module. In content coupling, modules are tightly intertwined. There is no semblance of independence. It is the worst type coupling. Fortunately, most high level languages do not have provision for creating content coupling.

13. Write down the advantages of functional independence.

By the term functional independence, we mean that a module performs a single task and needs very little interaction with other modules. A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules. Functional independence is a key to any good design primarily due to the following advantages it offers—

1) Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules. Further, once a failure is detected, error isolation makes it very easy to locate the error.

2) Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program.

3) Understandability: When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other.

14. What is Unit Testing? Explain it briefly.

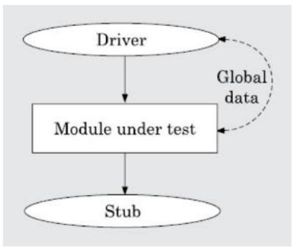
Unit testing is referred to as testing in the small. Unit testing is undertaken after a module has been coded and reviewed. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed.

1) <u>Driver and stub modules:</u> In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module—

a) The procedures belonging to other modules that the module under test calls.
b) Non-local data structures that the module accesses.
c) A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

i) <u>Stub:</u> The role of stub and driver modules is pictorially shown in the following figure—



A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table look up mechanism.

ii) <u>Driver:</u> A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.