Introduction to Software Engineering.

Software Engineering is an engineering technique branch related to the evolution of software product using well-defined scientific principles techniques, and procedures. The result of software engineering is an effective and reliable software product.

Software engineering is a systematic, disciplined quantifiable study and approach to the design development, operation and maintenance of a software system.

Software: The software is a collection of integrated programs. Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages.
Computer programs and related documentation such as requirements, design models and user manuals.

Types of Software

1) Application Software: This is end-user programs that help you perform tasks or achieve a desired outcome. Internet browsers, photo-editing software are some examples. Software that performs special functions or provides functions that are much more than the basic operation.

Features:

Performs more specialized tasks like word processing.
Requires more storage software
More Interactive
Easy-to design and understand.
High-level language.

Types

1) General Purpose Software: Used for variety of tasks and not limited to performing a specific task only. Ex- MS-word / MS-Excel.

2) Customized Software: Used to perform specific task or designed for specific organisations Ex- Railway, Invoice management.

3) Utility Software: Used to support the comp. infrastructure. To analyze, configure, optimize and take care of system. Ex- disk repair, antivirus

2) System Software: It directly operates the comp hardware and provides the basic functionality to the users as well as to other software to operate smoothly. Controls a computer's internal functioning and hardware devices as monitor, storage devices etc. It helps user applications to to communicate with hardware.

Features:
Closer to computer system.
Low-level language.
Difficult to design.
Fast in working speed.
Less Interactive.

1) Operating System: The first software that loads into computer's memory. Manages all resources such as memory, CPU, printer, hard disk, etc. Ex- Linux, Apple macOS, Microsoft Windows etc

2) Language Processor: It converts programs written in high-level programming languages like Java, C++ into set of instructions that are easily readable by machines.

3) Device Driver: A device driver is a program that controls a device and helps that device to perform its functions. It knows how to control or manage that device.

IMPORTANCE OF SOFTWARE ENGINEERING

1) Reducas Complexity: Big software is always complicated and challenging to progress. It divides big problems into various small issues. And then start solving each small issue one by one. Then solved independently.

2) Minimize Software Cost: Software needs a lot of hardwork and software engineers are highly paid experts. A lot of manpower is required to develop software with a large number of codes. But in software engineering, programmers project everything and decrease all those things that are not needed. In turn the cost for software productions becomes less as compared to any software that does not use this method.

3) To decrease Time: Anything that is not made according to the project always wastes time. And if you are making great software, then you may need to run many codes to get the definitive running code. So if you are making your software according to the software engineering method, then it decreases time.

Page:

4) Handling Big Projects: To handle a big project without any problem, the company must go for software engineering method.

5) Reliable Software: Software should be secure. And if any bugs come in the software, the company is responsible for solving all these bugs. Because in software engineering, testing and maintenance are given, so there is no worry of its reality. reliability.

6) Effectiveness: Effectiveness comes if anything has made according to the standards. Software standard are the big target of companies to make it more effective. So, software becomes more effective in oct with help of software engineering.

OBJECTIVES / SOFTWARE CHARACTERISTICS

1) Maintainability
2) Efficiency.
3) Correctness.
4) Reusability.
5) Testability.
6) Reliability.
7) Portability.
8) Adaptability.
9) Interoperability.

Date:
Page:

NEED OF SOFTWARE ENGINEERING

The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.

1) Huge Programming: It is simpler to manufacture a wall than to a house or building, similarly, as the measure of programming become extensive engineering has to step to give it a scientific process.

2) Adaptability: If the software procedure were not based on scientific and engineering ideas, it would be simpler to recreate new software than to scale an existing one.

3) Cost: As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the paper process not adapted.

4) Quality Management: Better procedure of software development provides a better & quality software product.

Date:
Page:

5) Dynamic Nature: The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one

SOFTWARE REQUIREMENT SPECIFICATION

It accomodates life cycle evolution from a growth and requirement change.
It incorporates software quality objective into the product. It focuses on early error deduction.

Requirement Engineering Process.

The first step in it is to gain an understanding of the problem for which the software is being developed for - this all necessary info about the problem are gathered. Based on the info the initial requirement specification of the software is developed. The initial specification is further refined after discussion with the perspective user and stake holder

Types of Software Requirement

1) **Design Requirement**: It specifies the design of the system or system component.

2) **Functional Requirement**: It specifies the function that a system or system component must perform.

3) **Implementation Requirement**: It specifies the coding or construction of a system or system component. This affected the technical effectiveness.

4) **Interface Requirement**: It specify the external items with which the system or its component must interact and it is and the construction on format, timing or other factors calls by such interaction.

5) **Performance Requirement**: It imposes conditions on a function requirement for e.g. a requirement that specify the speed, accuracy or memory uses with which a given function must be perform.

6) **Physical Requirement**: It specify a physical characteristics that a system or system component must be developed.

Structure of SRS (Software Requirement Specification)

SRS is a final work product produced by the requirement engineer. It serves as the foundation for the subsequent foundation software engineering activities. It discribes the function and performance requirement of the software. It also lists the constraint that will affect its development.

SRS is a formal document. It uses natural language, graphical representations, mathematical models, scenarios, prototype model or any combination of these to discribe the software to be developed. There are standard templates for presenting requirement specification in a consistent and more understandable manner.

The characteristics of good SRS documents are as follows:

a) **Structure**: Should be well structured. A well structured document is easy to understand and modify.

b) **Concise**: Should be concised consistent & complete.

c) **Conceptual Integrity**: Should have conceptual integrity so that it can be easily understood by reader without any ambiguity.

d) **Viewed as black box**: should only specify the functions that the software is required to perform and not how it should do the required functions. Hence, the SRS document views the software as blackbox. Also called the black box specification of a system.

e) **Response to Exceptional conditions**: Should list all conceivable exceptional conditions or events and specify software response to such conditions.

f) **Verifiable**: All requirement of the software as document should be verifiable. It should be possible to determine whether the requirement stated in the SRS document have been met in implementation or not.

Software Metrics

Metric is quantity measure of the degree to which a given attribute is processed by a system or its component or by a process. Software metric are measures that are used to quantify different attributes of software program.
Software metric estimation and measure of different attribute of software and software developm process.
Software metrices can be classified into 3 types:

1. Product- Metrics
2. Process Metrics
3. Resource Metrics

1. The measures of different characteristics of software product. The two important characteristics are:
   a) Size & Complexity
   b) Quality & Reliability

2. The measures of different characteristics of software development process. They quantify different aspect of the process being used to develop the software.

3. Quantify measures of various resources used in software project- Software projects uses 3 major types of resources
Human Resource
Hardware Resource
Software Resource
It denotes how efficiently the available resources are in used in project.

Software Size Metrics

Most of initial work in product metrics is dealt with characteristics of source code. The most important is the size of software.
A no. of different software size metrics have been proposed & used. Size is typically a direct count of selected characteristics to disaibe volume of software product- Can be expressed in 3 following ways.

a) In Terms of Physical Size of Program.
b) In Terms of meaningful functions/services it provides.
c) In Terms of Logical size

a) Lines of Code Metrics: Traditionally the LOC or KLOC is the primary measure of software size. Most lines of code metrics count all executable instructions and data declarations but exclude comments, blanks & continuation lines. Estimate size through analogy by comparing new software to similar functions found in other existing application.
Having more detailed info about functionality of new software clearly provides basics for better comparison

Function Point Metrics

Was developed in IBM in the year 1977 func. point of application are determined by counting number & types of function used in the application. Various functions used in application can be categorized into five types

Category	Property
1. External Input	All unique data of control input that cross the system boundary and cause processing to occur.
2. External Output	All unique data of control output that cross the system boundary after the processing has occurred.

3. External Inquiry All unique transaction that cross the system boundary to make active demand on system.

4. Internal File All logical of data that are stored within a system according to some predefined conceptual schema

5. External Interface All unique files or program that cross system boundary & are shared with atleast one other system or application

Constructive or Estimation Model

Was proposed by Boehm in 1981 and as the model for estimating effort, cost, and schedule for s/w projects.

The estimation of s/w projects are done in 3 stages / models:

1. **Basic COCOMO**

Gives an approximate estimate of effort for software project- based on a single attribute and the size of the software expressed in terms of KLOC is the most important attribute for estimation of effort

According to BASIC COCOMO, the expressions of estimation of efforts required for s/w development is

$$Effort = A \times (size)^B$$

where A & B are constants.

Effort estimation is expressed in units of PMs. The amount of effort estimated for s/w development is main

Date:
Page:

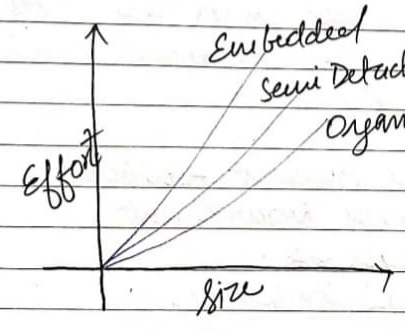
parameter for estimations of time.

Expressions for estimation of time required for s/w development is;

$$T = C \times (Effort)^D$$

where C & D are constants.

Types of Software	A	B	C	D
Organic	2.4	1.05	2.5	0.38
Semi detached	3.0	1.12	2.5	0.38
Embedded	3.6	1.20	2.5	0.38



2. Intermediate COCOMO

The purpose of basic COCOMO is to get rough estimate of effort and time required for software development. It considers only one attribute of software, i.e., software size expressed in KLOC. However, beside the software size, there are various other factors that affect the amount of effort & time duration needed for completion of software project.

The intermediate COCOMO recognizes 15 factors / cost drivers that affect the software project. These 15 factors can be categorized into 4 types

1. Software Product Attributes
   - 1.1 Reliability Requirement
   - 1.2 Database Size.
   - 1.3 Product Complexity.

2. Computer System Attributes
   - 2.1 Execution Time Constraints.
   - 2.2 Main Storage Constraints.
   - 2.3 Virtual Machine Volatility.
   - 2.4 Computer Turn around Time.

3. Human Resource Attribute
   - 3.1 Analyst Capability
   - 3.2 Virtual Machine Experience
   - 3.3 Programmer capability

3.4 Programming Language Experience
3.5 Application Experience.

4. Software Product Attributes.
4.1 Use of Modern Practices
4.2 Use of Software tools.
4.3 Required Development Schedule.

3. Complete COCOMO
Any large software system will consist of different components & sub systems. The complexity of these components may not be the same. For example, some subsystem may be too simple whereas some may be quite complex. Hence, some of the sub systems of large system may be considered as organic, as semi-detached, as embedded type
Basic and Intermediate COCOMO are considered as software project as single homogeneous entity, however, complete COCOMO considers system to be heterogeneous in nature to apply complete COCOMO. The software is broken down into subsystem. The sub system are categorized into organic, semi-detached or embedded type. The sub systems are estimated separately and then added up to get the estimates of complete project.

FREEMIND

Date

Page

S/w life cycle models :-

Classical waterfall Model :- divides the life cycle into a set of phases. This model considers that one phase can be started after completion of the previous phase. That is the output of one phase will be the input to the next phase. The development process can be considered as a sequential flow in the waterfall.

* Sequential phases of Waterfall Model :-

Feasibility Study
↓
req. analysis and specification
↓
design
↓
coding and unit testing
↓
maintenance

→ Adv. of Classical Waterfall Model :-

- This model is very simple and is easy to understand.
- Phases in this model are processed one at a time.
- Each stage in the model is clearly defined
- This model has very clear and well understood milestones
- Process, actions and results are very well documented

→ Drawbacks of classical waterfall model :-

→ classical waterfall model can't be used in real projects, but instead we use other SDLC models which are based on the classical waterfall model.

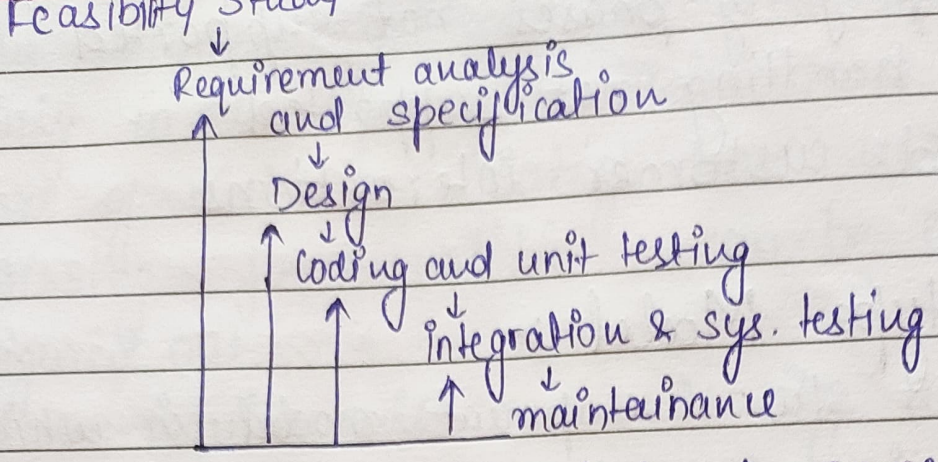
→ No feedback path :- it assumes that no error is ever committed by developers during any phases. ∴ it doesn't incorporate any mechanism for error connection.

→ difficult to accommodate change request :- it is diffi. to change request after the requirements specification is complete. as this model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but actually customers req. keep changing.

→ No overlapping of phases :- new phase can only be started when the previous phase is completed.

Iterative Waterfall Model :- is almost same as the classical WFM except some changes are made to ↑ the efficiency of the s/w development.

- The iterative waterfall model provides feedback paths from every phase to its preceding phases, which is the main diff. from the classical waterfall model.



When errors are detected at some later phase these feedback paths allow correcting errors committed by programmers during some phase The feedback path allows rework, but there is no feedback path to stage - feasibility study bc. once the project has been taken, doesn't give up easily.

Phase Containment of Errors :- The principle of detecting errors as close to their pts. of commitment as possible is known as...

Adv. of IWFM :-

→ feedback path
→ simple (it is widely used s/w dev. models)

Drawbacks of IWFM :-

major → difficult to incorporate change request :- all the requirements must be clearly stated before starting of the dev. phase.
→ incremental delivery not supported :- there is no scope for any intermediate delivery.

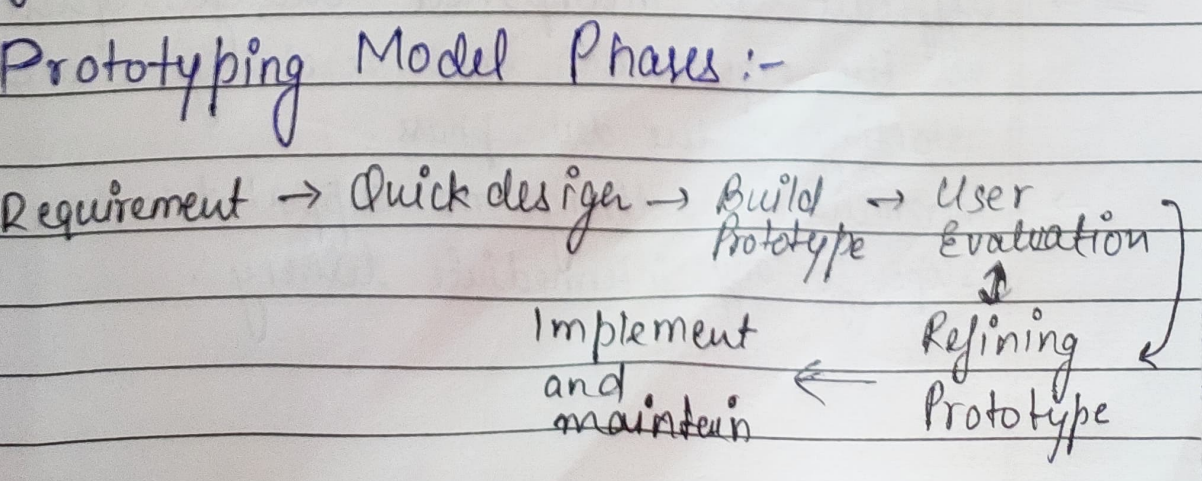
Date
Page

→ Overlapping of phases not supported
→ risk handling not supported
→ limited customer interactions

Prototype Model :-

is a s/w development model in which proto-type is built, tested and reworked until an acceptable prototype is achieved.
It works best in scenarios where the project's requirements are not known in detail. It is an iterative, trial and error method which takes place b/w developer and client. In this model, a prototype of the end product is first developed, tested and refined as per customer feedback repeatedly till a final acceptable prototype is achieved.
The system is partially implemented before or during the analysis phase thereby giving the customers an opportunity to see the product early in the life cycle.

Prototyping Model Phases :-



Advantages :-

→ The customers get to see the partial product early in the life cycle. This ensures a grtr. level of customer satisfaction and comfort

→ Missing functionalities can be easily figured out.

→ New requirements can be easily accomodated Flexibility in design

Disadvantages :-

→ Poor documentation due to continuosly changing customer requirements.

→ There may be too much variation in requirements each time the prototype is evaluated by the customer.

→ It is very difficult for developers to accomodate all the changes demanded by the customer

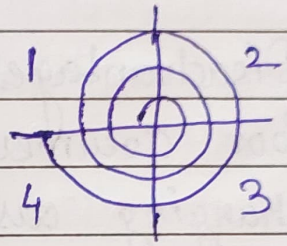
FREEMIND

Date

Page

Spiral Model :-

S.M ~~is one of~~ provides support for risk handling. It's diagrammatic representation looks like a spiral with many loops. The exact no. of loops of the spiral is unknown and can vary from project to project. Each loop of the spiral is called a Phase of the s/w dev. process.



the radius of the spiral at any point represents the expenses (cost) of the project so far
Angular dimension represents the process made so far in the current phase.

1. Objectives determination and identify alt. sol^{n}. :- Requirements are gathered, ~~from the customers~~ objectives are identified, elaborated, and analyzed at the start of every phase. Then, all the possible sol^{n}. are proposed.
2. Identify and resolve risk : ....
3. Develop next version of the product....
4. Review and plan for the next phase....

Risk Handling in Spiral Model: a risk is any situation that may effect successful completion of any project.
R.H is the best feature of spiral Model

Date
Page

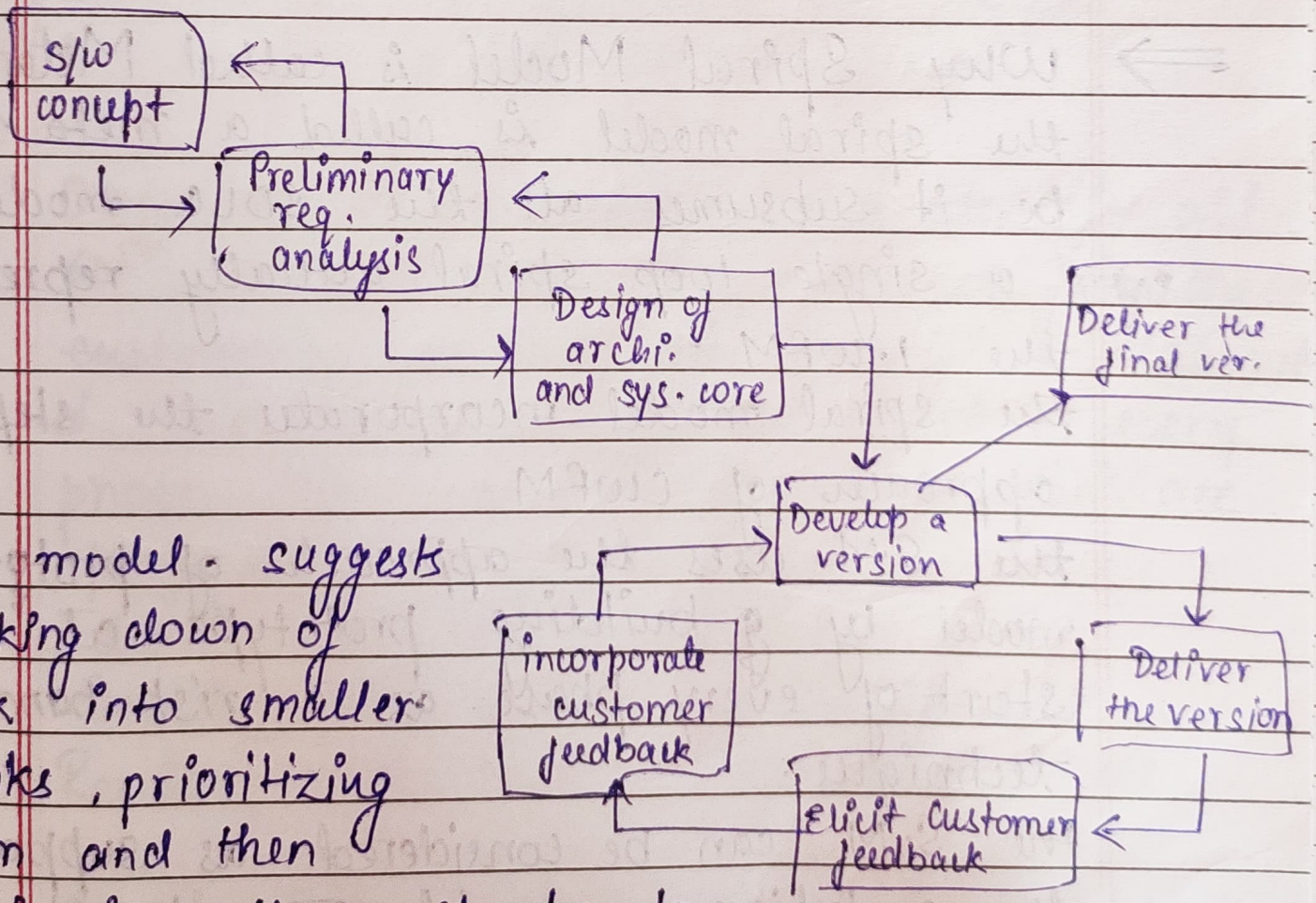
as you can handle the unknown risks ~~even~~ after the project has started. these risk resolutions are done by developing a prototype. The prototype can be build on every phase of s/w dev. in spiral model

The Prototype Model also supports risk handling but there we've to identify the risk completely before the start of the dev. work of the project.
But irl ~~risk~~ project risk may occur at any time after the dev. work has started

⟹ Why Spiral Model is called Meta Model
the spiral model is called a meta model bc. it subsumes all the SDLC models
e.g., a single loop spiral actually represents the 1.WFM
the spiral model incorporates the stepwise approach of CWFM
the S.M uses the approach of prototyping model by building prototype at the start of every phase as a risk handling technique.
the S.M can be considered as supporting the evolutionary model. The iterations along the spiral can be considered as evolutionary levels through which the complete system is built.

Adv.	Disadv.
→ Risk Handling	→ Complex
→ Good for large projects	→ Expensive (for small projects)
→ Flexibility in requirements	→ Too much dependability on risk analysis
→ customer satisfaction	→ Difficulty in time management

Evolutionary Model :- All the models hv. the disadv. that the duration of time from start of the project to the delivery time of a solution is very high. E.M solves this problem.



Evol. model. suggests breaking down of work into smaller chunks, prioritizing them and then delivering those chunks to the customer one by one.

Page

Application of E.M :-

1. it is used in large projects where you can easily find modules for incremental implementation. E.M is commonly used when the customer wants to start using the core features instead of waiting for the full s/w.
2. E.M is used in OOSD (obj. ori. s/w dev) bez the sys. can be easily portioned into units in terms of objects.

Adv. :-

→ the user gets a chance to experiment partially developed sys.

→ It reduces the error bcz. the core modules gets tested thoroughly.

Disadv. :-

→ sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.

S/w re-eng. :- is reorganizing and modifying existing s/w sys. to make them more maintainable

Steps :- ① Inventory Analysis's → Doc. reconstruction → Reverse eng. ← code " ← data reconst. ← Fwd eng.

Requirement Analysis :-

also known as req. eng., is the process of defining user expectations for a new s/w being built or modified

~~Steps :-~~

~~Draw the context Diagram~~

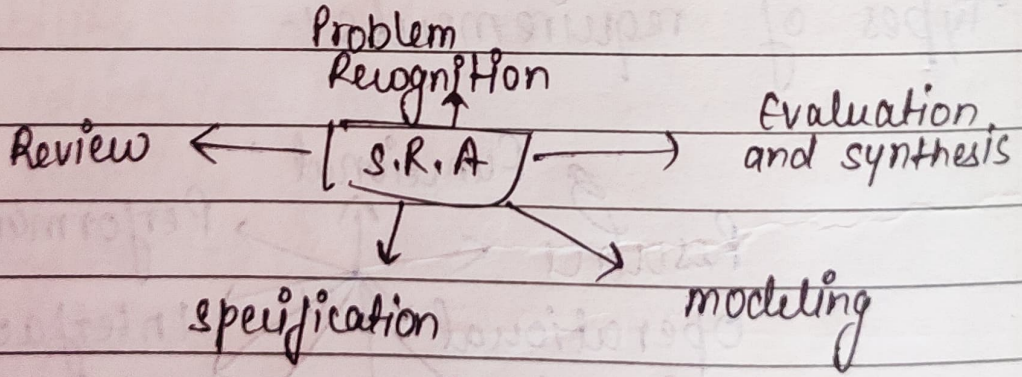
~~Develop Prototypes~~

~~Model the requirements~~

~~Finalise the requirements~~

Requirements are generally a type of expectation of user from s/w product that is imp. and need to fulfilled by s/w.

Analysis means to examine something in an organized and specific manner to know complete details about it.



Feasibility Study :- is the feasibility-analysis or it is a measure of the s/w product in terms of how much beneficial product development will be for the organisation in a practical point of view.

Boehm's definition of organic, semidetached & embedded systems :-

Organic :- A s/w project is said to be an organic type if the team size req. is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.

Semi-detached :- A s/w project is said to be a semi-detached type if the vital charact.

mer.
much
ne
of the sy

such as team-size, exp., knowledge of various programming env. lie in b/w that of organic and embedded. The projects classified as semi-detached are comparatively less familiar and difficult to develop compared to the organic one and require more exp. and better guidance and creativity.

Embedded :- A s/w project with requiring the highest level of complexity, creativity and exp. requirement fall under this category. Such s/w requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

Size Oriented Metrics :-

LOC Metrics :-

→ is considered to be the normalization value.

→ productivity is defined as KLOC / Effort, where effort is measured in person-months

→ size-oriented metrics depend on the programming used.

→ as productivity depends on KLOC :. assembly lng. have more prod.

Disadvantage :-

→ It is defined on the code

→ It is lng. dependent

→ bad s/w ~~code~~ design may cause an excessive line of code

→ it takes no acct. of functionality / complexity

⟹ The problems that an organization will face if it doesn't not using a SRS document are as follows :-

- without developing the srs document, the system would not be implemented according to customer needs.
- s/w developers would not know whether what they're developing is what exactly required by the customer.
- without SRS doc. it will be very much diffi. for the ~~maintain~~ maintenance engineers of understand the functionality of the sys

- It will be very much difficult for user doc. writers to write the users manuals properly without understanding the SRS doc.

Software Design :- is a mechanism to transform user requirements into some suitable form which helps the programmer in s/w coding and implementation.

Objectives / Purpose of s/w design :-
Correctness, Completeness, Efficiency, Flexibility, Consistency, Maintainability.

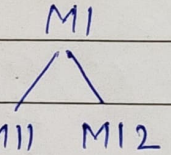
Coupling :- is the degree of interdependence b/w s/w modules. A good design is the one that has low coupling. Coupling is measured by the no. of relations b/w the modules i.e., the coupling ↑ as the calls b/w modules ↑ or the amt. of shared data is large.
A design with high coupling will have more errors

Types of coupling :-

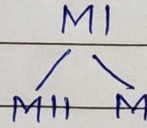
no direct, data, stamp, control, external, common, content

worst ——————————→ Best

No direct :- there is no direct coupling b/w M1 & M2
modules are subordinates to diff. modules.



Data Coupling :- when data of one module is passed to another module.

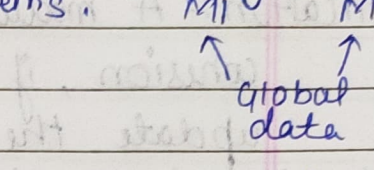


Stamp Coupling :- if 2 module are stamp coupled if they communicate using composite data items such as structure, objects etc. when the module passes non-global data structure or entire structure to another module.

Control Coupling :- exists among 2 modules if data from 1 module is used to direct the structure of instruction execution in another.

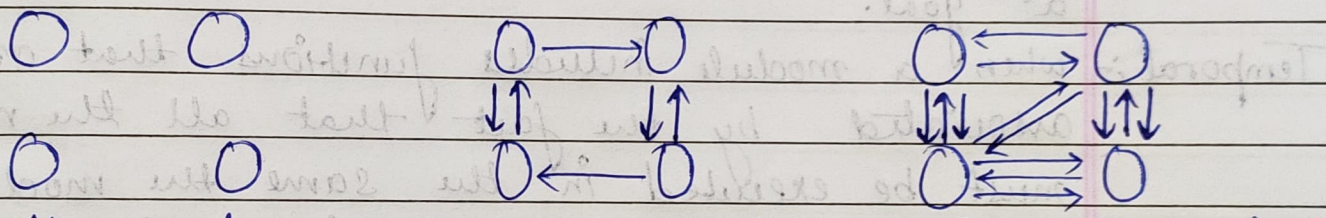
External :- arises when 2 modules share an externally imposed data format, communication protocols or device interface

Common :- Two modules are common coupled if they share info. through some global data items.



Content :- exists among two modules if they share code e.g., a branch from one module into another module.

Coupling techniques :-



Cohesion :- defines to the degree to which the elements of a module belong together. It measures the strength of relationships b/w pieces of functionality within a given module.

Types of cohesion :-

Functional > Sequential, Communication, Procedural, Temporal, Logical, Coincidental

Best ——————→ Worst

Functional Cohesion :- is said to exist if the diff. elements of a module cooperate to achieve a single function.

Sequential :- A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next

Communication :- A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure.

Procedural :- A module is said to be procedural cohesion if the set of the purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal.

Temporal :- when a module includes functions that are associated by the fact that all the methods must be executed in the same the module is said to exhibit temporal cohesion.

Logical :- if all the elements of the module perform a similar operation.

Coincidental :- if the module performs a set of tasks that are associated with each other very loosely

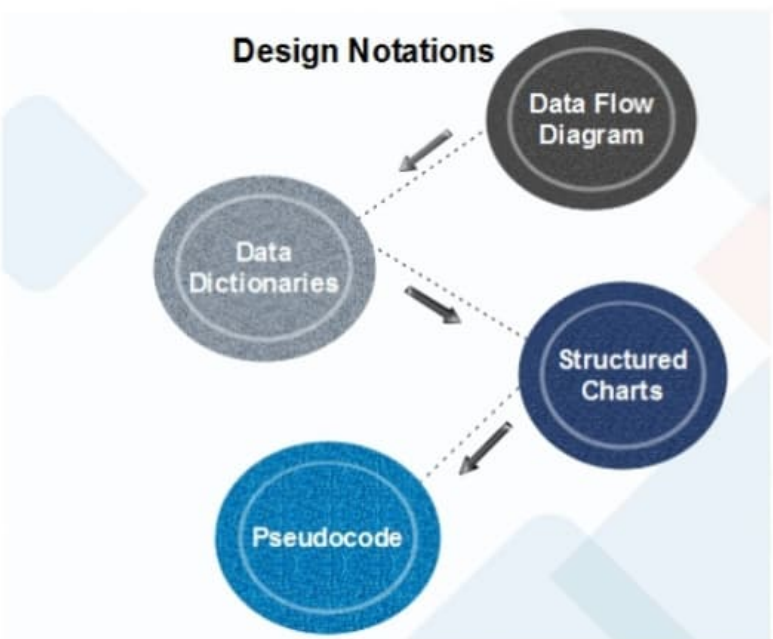
Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:



Data Flow Diagram

Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs. The design is described as data-flow diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

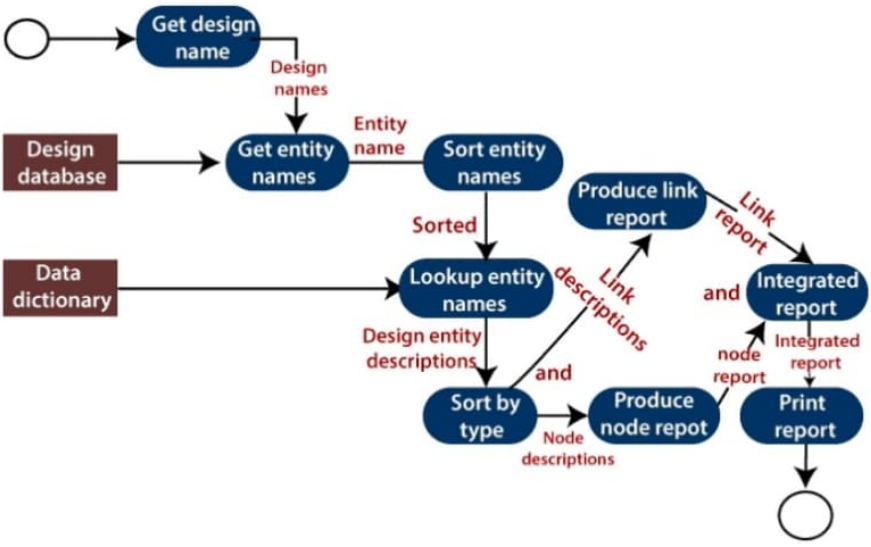
Data-flow diagrams are a useful and intuitive way of describing a system. They are generally understandable without specialized training, notably if control information is excluded. They show end-to-end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

Data-flow design is an integral part of several design methods, and most CASE tools support data-flow diagram creation. Different ways may use different icons to represent data-flow diagram entities, but their meanings are similar.

The notation which is used is based on the following symbols:

Symbol	Name	Meaning
	Rounded Rectangle	It represents functions which transforms input to output. The transformation name indicates its function.
	Rectangle	It represents data stores. Again, they should give a descriptive name.
	Circle	It represents user interactions with the system that provides input or receives output.
	Arrows	It shows the direction of data flow. Their name describes the data flowing along the path.
"and" and "or"	Keywords	The keywords "and" and "or". These have their usual meanings in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

Data flow diagram of a design report generator



The report generator produces a report which describes all of the named entities in a data-flow diagram. The user inputs the name of the design represented by the diagram. The report generator then finds all the names used in the data-flow diagram. It looks up a data dictionary and retrieves information about each name. This is then collated into a report which is output by the system.

Data Dictionaries

A data dictionary lists all data elements appearing in the DFD model of a system. The data items listed contain all data flows and the contents of all data stores looking on the DFDs in the DFD model of a system.

A data dictionary lists the objective of all data items and the definition of all composite data elements in terms of their component data items. For example, a data dictionary entry may contain that the data *grossPay* consists of the parts *regularPay* and *overtimePay*.

grossPay = regularPay + overtimePay

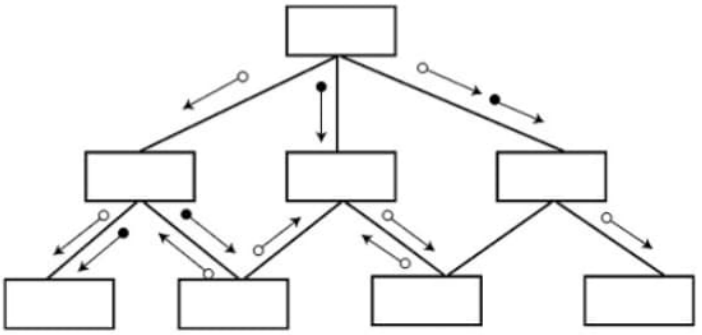
For the smallest units of data elements, the data dictionary lists their name and their type.

A data dictionary plays a significant role in any software development process because of the following reasons:

- A Data dictionary provides a standard language for all relevant information for use by engineers working in a project. A consistent vocabulary for data items is essential since, in large projects, different engineers of the project tend to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary provides the analyst with a means to determine the definition of various data structures in terms of their component elements.

Structured Charts

It partitions a system into block boxes. A Black box system that functionality is known to the user without the knowledge of internal design.

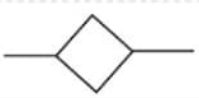


Hierarchical format of a structure chart

Structured Chart is a graphical representation which shows:

- System partitions into modules
- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

The following notations are used in structured chart:

SYMBOL	DESCRIPTION
	Module
	Arrow
	Data couple
	Control Flag
	Loop
	Decision

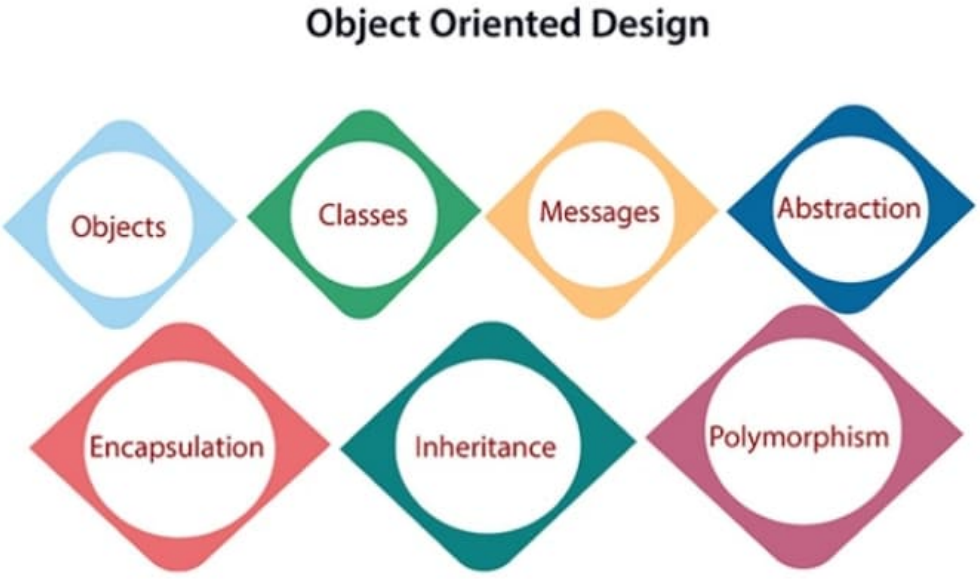
Pseudo-code

Pseudo-code notations can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise, English Language phases that are structured by keywords such as If-Then-Else, While-Do, and End.

Object-Oriented Design

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

The different terms related to object design are:



1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with 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. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Goals of Coding

1. **To translate the design of system into a computer language format:** The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.
2. **To reduce the cost of later phases:** The cost of testing and maintenance can be significantly reduced with efficient coding.
3. **Making the program more readable:** Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

Characteristics of Programming Language

Readability: A good high-level language will allow programs to be written in some methods that resemble a quite-English description of the underlying functions. The coding may be done in an essentially self-documenting way.

Portability: High-level languages, being virtually machine-independent, should be easy to develop portable software.

Generality: Most high-level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.

Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs mean in high-level languages are often significantly shorter than their low-level equivalents.

Error checking: A programmer is likely to make many errors in the development of a computer program. Many high-level languages invoke a lot of bugs checking both at compile-time and run-time.

Cost: The ultimate cost of a programming language is a task of many of its characteristics.

Quick translation: It should permit quick translation.

Efficiency: It should authorize the creation of an efficient object code.

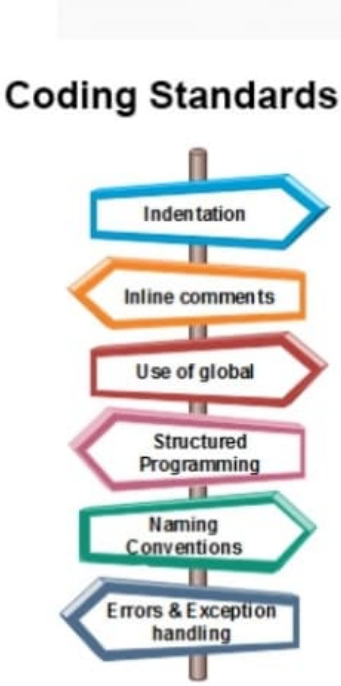
Modularity: It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self-consistency among these modules.

Widely available: Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

Coding Standards

General coding standards refers to how the developer writes code, so here we will discuss some essential standards regardless of the programming language being used.

The following are some representative coding standards:

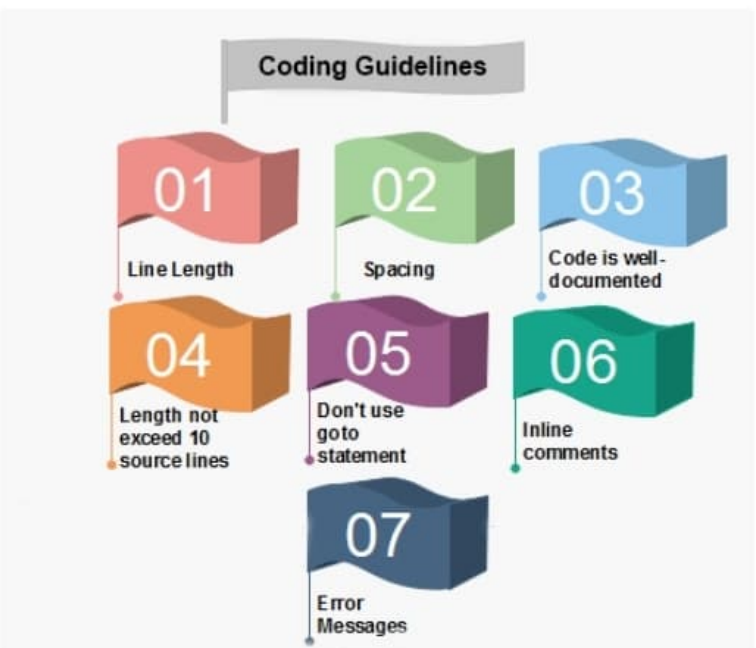


1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs.
   Indentation should be used to:
   - Emphasize the body of a control structure such as a loop or a select statement.
   - Emphasize the body of a conditional statement
   - Emphasize a new scope block
2. **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
3. **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.
4. **Structured Programming:** Structured (or Modular) Programming methods shall be used. "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.
5. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
6. **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

Coding Guidelines

General coding guidelines provide the programmer with a set of the best methods which can be used to make programs more comfortable to read and maintain. Most of the examples use the C language syntax, but the guidelines can be tested to all languages.

The following are some representative coding guidelines recommended by many software development organizations.



1. Line Length: It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

2. Spacing: The appropriate use of spaces within a line of code can improve readability.

Example:

Bad:
```
cost=price+(price*sales_tax)
fprintf(stdout ,"The total cost is %5.2f\n",cost);
```

Better:
```
cost = price + ( price * sales_tax )
fprintf (stdout,"The total cost is %5.2f\n",cost);
```

3. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

4. The length of any function should not exceed 10 source lines: A very lengthy function is generally very difficult to understand as it possibly carries out many various functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.

5. Do not use goto statements: Use of goto statements makes a program unstructured and very tough to understand.

6. Inline Comments: Inline comments promote readability.

7. Error Messages: Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

Difference between Inspection and Walkthrough

1. Walkthrough :

Walkthrough is a method of conducting informal group/individual review. In a walkthrough, author describes and explain work product in a informal meeting to his peers or supervisor to get feedback. Here, validity of the proposed solution for work product is checked.

It is cheaper to make changes when design is on the paper rather than at time of conversion. Walkthrough is a static method of quality assurance. Walkthrough are informal meetings but with purpose.

2. Inspection :

An inspection is defined as formal, rigorous, in depth group review designed to identify problems as close to their point of origin as possible. Inspections improve reliability, availability, and maintainability of software product.

Anything readable that is produced during the software development can be inspected. Inspections can be combined with structured, systematic testing to provide a powerful tool for creating defect-free programs.

Inspection activity follows a specified process and participants play well-defined roles.
An inspection team consists of three to eight members who plays roles of moderator, author, reader, recorder and inspector.

For example, designer can acts as inspector during code inspections while a quality assurance representative can act as standard enforcer.

Stages in the inspections process :

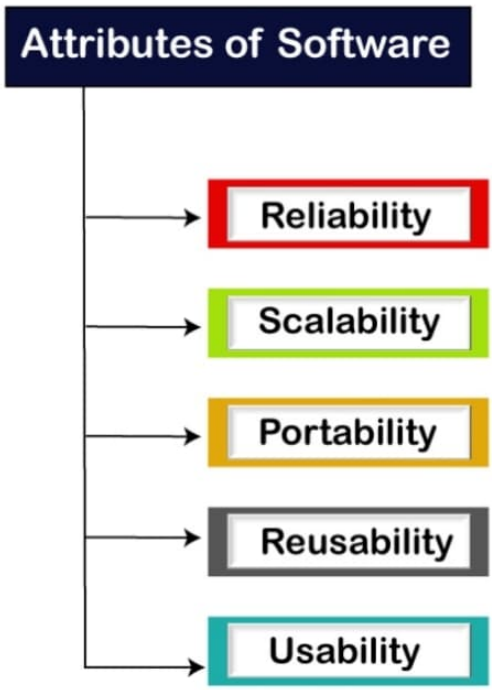
- **Planning :** Inspection is planned by moderator.
- **Overview meeting :** Author describes background of work product.
- **Preparation :** Each inspector examines work product to identify possible defects.
- **Inspection meeting :** During this meeting, reader reads through work product, part by part and inspectors points out the defects for every part.
- **Rework :** Author makes changes to work product according to action plans from the inspection meeting.
- **Follow-up :** Changes made by author are checked to make sure that everything is correct.

Difference between Inspection and Walkthrough :

S.No.	Inspection	Walkthrough
1.	It is formal.	It is informal.
2.	Initiated by project team.	Initiated by author.
3.	A group of relevant persons from different departments participate in the inspection.	Usually team members of the same project take participation in the walkthrough. Author himself acts walkthrough leader.
4.	Checklist is used to find faults.	No checklist is used in the walkthrough.
5.	Inspection processes includes overview, preparation, inspection, and rework and follow up.	Walkthrough process includes overview, little or no preparation, little or no preparation examination (actual walkthrough meeting), and rework and follow up.
6.	Formalized procedure in each step.	No formalized procedure in the steps.
7.	Inspection takes longer time as list of items in checklist is tracked to completion.	Shorter time is spent on walkthrough as there is no formal checklist used to evaluate program.
8.	Planned meeting with the fixed roles assigned to all the members involved.	Unplanned
9.	Reader reads product code. Everyone inspects it and comes up with detects.	Author reads product code and his teammate comes up with the defects or suggestions.
10.	Recorder records the defects.	Author make a note of defects and suggestions offered by teammate.
11.	Moderator has a role that moderator making sure that the discussions proceed on the productive lines.	Informal, so there is no moderator.

What is Software Testing

Software testing is a process of identifying the correctness of software by considering its all attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.

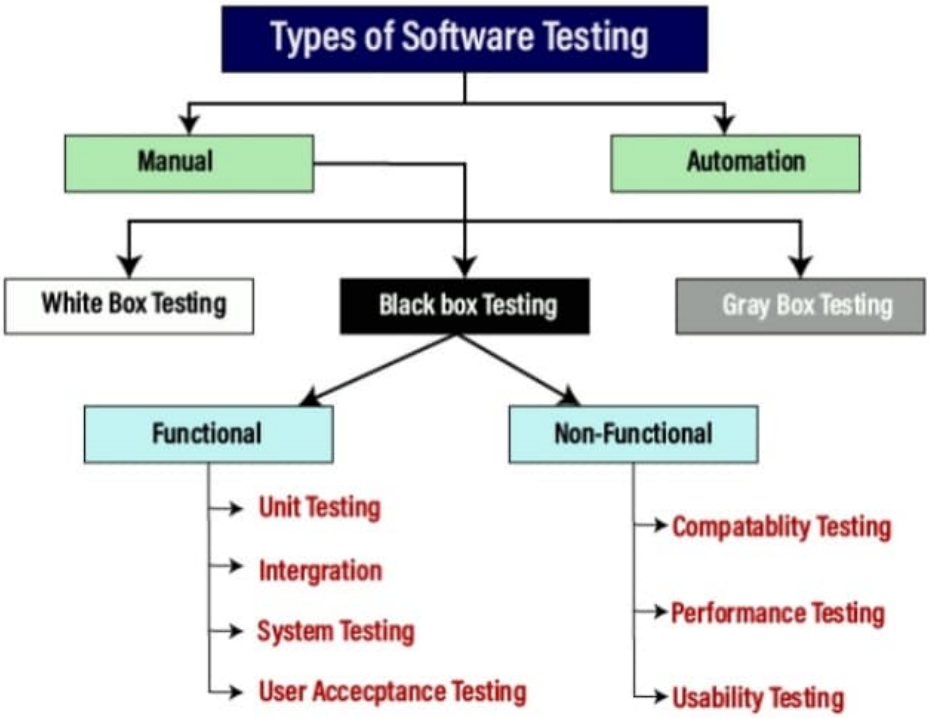


Software testing provides an independent view and objective of the software and gives surety of fitness of the software. It involves testing of all components under the required services to confirm that whether it is satisfying the specified requirements or not. The process is also providing the client with information about the quality of the software.

Type of Software testing

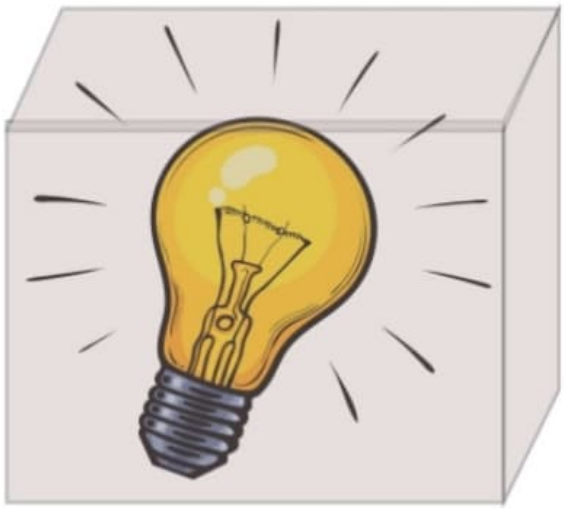
We have various types of testing available in the market, which are used to test the application or the software.

With the help of below image, we can easily understand the type of software testing:



White Box Testing

In white-box testing, the developer will inspect every line of code before handing it over to the testing team or the concerned test engineers.

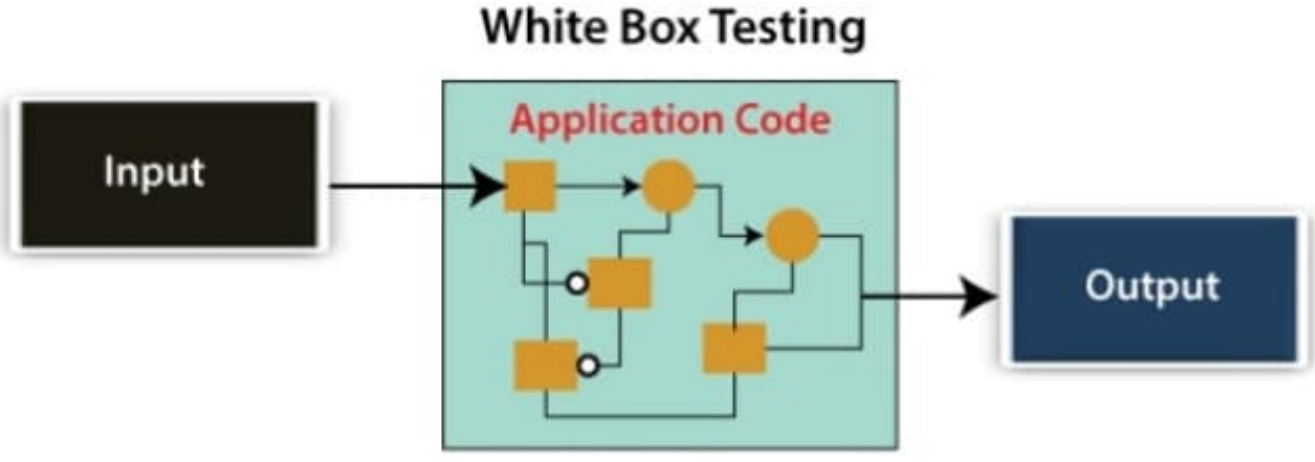


White Box Testing

Subsequently, the code is noticeable for developers throughout testing; that's why this process is known as **WBT (White Box Testing)**.

In other words, we can say that the **developer** will execute the complete white-box testing for the particular software and send the specific application to the testing team.

The purpose of implementing the white box testing is to emphasize the flow of inputs and outputs over the software and enhance the security of an application.



White box testing is also known as **open box testing, glass box testing, structural testing, clear box testing, and transparent box testing**.

Black Box Testing

Another type of manual testing is **black-box testing**. In this testing, the test engineer will analyze the software against requirements, identify the defects or bug, and sends it back to the development team.

Black Box Testing

Then, the developers will fix those defects, do one round of White box testing, and send it to the testing team.

Here, fixing the bugs means the defect is resolved, and the particular feature is working according to the given requirement.

The main objective of implementing the black box testing is to specify the business needs or the customer's requirements.

Here, fixing the bugs means the defect is resolved, and the particular feature is working according to the given requirement.

The main objective of implementing the black box testing is to specify the business needs or the customer's requirements.

In other words, we can say that black box testing is a process of checking the functionality of an application as per the customer requirement. The source code is not visible in this testing; that's why it is known as **black-box testing**.

1. Unit Testing

Unit testing is the first level of functional testing in order to test any software. In this, the test engineer will test the module of an application independently or test all the module functionality is called **unit testing**.

The primary objective of executing the unit testing is to confirm the unit components with their performance. Here, a unit is defined as a single testable function of a software or an application. And it is verified throughout the specified application development phase.

2. Integration Testing

Once we are successfully implementing the unit testing, we will go integration testing. It is the second level of functional testing, where we test the data flow between dependent modules or interface between two features is called **integration testing**.

The purpose of executing the integration testing is to test the statement's accuracy between each module.

3. System Testing

Whenever we are done with the unit and integration testing, we can proceed with the system testing.

In system testing, the test environment is parallel to the production environment. It is also known as **end-to-end** testing.

In this type of testing, we will undergo each attribute of the software and test if the end feature works according to the business requirement. And analysis the software product as a complete system.

User Acceptance Testing

The User acceptance testing (UAT) is done by the individual team known as domain expert/customer or the client. And knowing the application before accepting the final product is called as **user acceptance testing**.

In user acceptance testing, we analyze the business scenarios, and real-time scenarios on the distinct environment called the **UAT environment**. In this testing, we will test the application before UAI for customer approval.

Software Maintenance

Software maintenance is a part of the Software Development Life Cycle. Its primary goal is to modify and update software application after delivery to correct errors and to improve performance. Software is a model of the real world. When the real world changes, the software require alteration wherever possible.

Software Maintenance is an inclusive activity that includes error corrections, enhancement of capabilities, deletion of obsolete capabilities, and optimization.

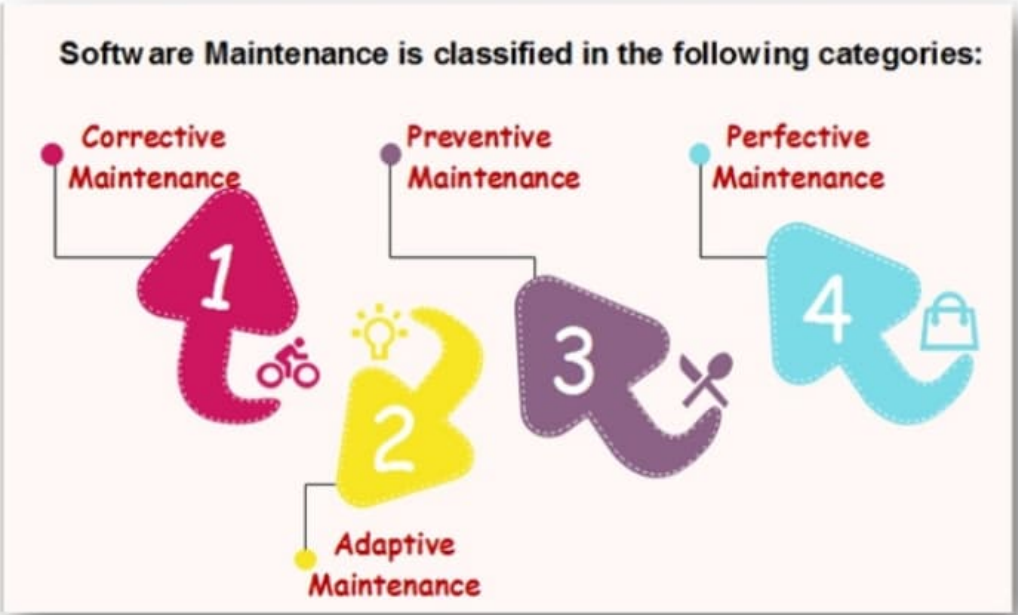
Need for Maintenance :

Software Maintenance is needed for:-

- Correct errors
- Change in user requirement with time
- Changing hardware/software requirements
- To improve system efficiency
- To optimize the code to run faster
- To modify the components
- To reduce any unwanted side effects.

Thus the maintenance is required to ensure that the system continues to satisfy user requirements.

Types of Software Maintenance



1. Corrective Maintenance

Corrective maintenance aims to correct any remaining errors regardless of where they may cause specifications, design, coding, testing, and documentation, etc.

2. Adaptive Maintenance

It contains modifying the software to match changes in the ever-changing environment.

3. Preventive Maintenance

It is the process by which we prevent our system from being obsolete. It involves the concept of reengineering & reverse engineering in which an old system with old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

4. Perfective Maintenance

It defines improving processing efficiency or performance or restricting the software to enhance changeability. This may contain enhancement of existing system functionality, improvement in computational efficiency, etc.