Software Engineering notes

Section - A

- Software engineering is a technique through which we can developed or created software for computer systems and any other electronic devices.
- In other words, Software engineering is a process in which user needs are analyzed and software are designed based on those needs.
- In Software engineering the development of software using well define scientific principle, method and procedures.
- Software engineers build these software and applications by using designing and programming languages.
- In order to create a complex software we should use software engineering technique as well as to reduce the complexity we should use abstraction and decomposition.
- Where abstraction describes only important part of software and leave irrelevant things for later stage of development. So, the requirement of software become simple.
- Whereas decomposition breakdowns the software into no of modules where each module produces as well define independent task.

The Problem domain :-

- The problem domain refers to the specific area of application or expertise that needs to be studied and understood in order to solve a problem using software.
- It focuses only on the relevant topics related to the problem and excludes everything else that is not required.
- In software development, the problem domain represents what the user wants to achieve such as business rules, user requirements, user stories, user use case.
- Understanding the problem domain is essential because incorrect or incomplete understanding leads to wrong requirements, poor design, rework, increased cost and project failure.

(i) Software engineering is much more than programming :-

- Software engineering is not limited to writing small programs.
- It involves design, development, documentation, testing, deployment and maintenance of software.
- It includes activities such as requirement analysis, system design, risk assessment, project management, documentation and team-work. Programming is the only one part of the overall software engineering process.

(ii) Software is expensive :-

- Software is more expensive compared to hardware because software development is a labor intensive activity that relies heavily on human skills and intellectual effort.
- It is difficult to develop the software within the prescribed budget due to complexity, requirement changes and the need for skilled professionals.

(iii) Software is late :-

- Software projects often fail to meet their delivery schedules. Estimating the required time and effort accurately is difficult because of changing requirements, lack of disciplined development process and unforeseen technical challenges.

(iv) Software brittleness :-

- Software brittleness refers to the tendency of software to fail when small changes are made.
- Since computers strictly follow instructions, even a minor modification or missing condition can cause unexpected failures.
- Poor design and lack of testing often make software brittle and hard to adapt to changes.

(v) Software is unreliable :-

- Software is considered unreliable when it does not behave according to customer requirements or fails under certain operating conditions.
- Inadequate requirements analysis, insufficient testing and poor design practices often lead to unreliable software systems.

Software Engineering Challenges :-

1. Maintaining Software Quality :-

- Maintaining software quality is one of the major challenges in software engineering. Software quality means conformance to explicitly stated functional and non functional requirements, documented development standards and implicit expectations such as usability, reliability and maintainability.
- Quality is measured against requirements, so poorly defined requirements result in poor quality software.

---

2. Changing Requirements :-

- Changing requirements are a constant challenge in software engineering. Customers often find it difficult to clearly express what they want until they see part of the system implemented.
- Requirements may change due to evolving business needs, new technologies, market competition or incomplete understanding of the problem domain during early stages of development.

---

3. Scalability :-

- Scalability is the ability of a software system to handle increasing workloads gracefully or to be expanded easily by adding resources.

- A scalable system continues to perform efficiently when the number of users, transactions, or data increases. Designing scalable software is difficult and requires careful architectural decisions.

---

4. Software is late :-

- Many software projects fails to meet deadlines due to inaccurate effort estimation, lack of disciplined processes, and frequent requirement changes.

---

5. Software is unreliable :-

- Software may be unreliable if it does not fully meet customer requirements or fails under certain conditions.
- Inadequate testing and poor design often contribute to unreliability.

---

6. Schedule Pressure :-

- Software project face intense schedule pressure due to business competitions and customer demands. Products are expected to be delivered quickly and updated frequently.
- This pressure often leads to aggressive commitments, reduced testing, compromised quality, and increased maintenance issues.

Software development life Cycle (Software engineering Approach)

(Section - A)

- SDLC stands for the Software development life Cycle.
- Software development life Cycle is a collection of process which are followed to develop a software.
- Software development life Cycle is a methodology that defines some process which are followed to develop a high quality software.
- It covers a detailed plan for building, deploying and maintaining the software.
- The main aim of SDLC is to define all the tasks required for developing and maintaining the software.
- It is followed for a software project within a software developing organization.

Requirement Analysis → Feasibility Study → Design → Coding → Testing → Deployment → Maintenance

---

Phases of Software development life Cycle :-

Phase - 1 Requirement Analysis :-

- It is the first phase of software development life Cycle in which all necessary information is collected from the customer to develop the software as per their expectation.
- Some important questions like: What is the need of the software, Who will be the end user, What is the future scope of that software are discussed.
- The main aim of this phase is to collect the details from the customer so that developers can easily understand what they are developing and how to fulfill the customer's requirement.
- This phase gives a clear picture of what we are going to build.

Phase - 2 Feasibility Study :-

- It is the second phase of the software development life Cycle in which an organisation discusses about the cost and benefits of the software.
- It is an important phase because profit from the software plays an important role as if cost is very high then company may face loss.

- After the feasibility study, the project may be accepted, accepted with modification or rejected.
- It measures how much beneficial the product is for the organisation.

Phase - 3 Design :-

- It is the third phase in which architects starts working on the logical designing of the software.
- In this SRS (System requirement specification) document is created which contains all logical details like how the software looks like, which programming language will be used, database design etc.
- This phase provides the prototype of the final product.
- Basically all it includes is design of everything which has to be coded.

Phase - 4 Coding :-

- When the designing of the software is completed, then a group of developers starts coding of the design using a programming language.
- The interface of the software and all its internal working according to the design phase is implemented in coding phase.
- A number of developers code the modules and then all modules are arranged together to work efficiently.
- It is the longest phase of software development life Cycle.

Phase - 5 Testing :-

- Once the software is completed then it is sent to the testers. Then the testing team starts testing the functionality of the entire system.
- In this phase software is checked for bugs or errors. If a bug is found then the software is resent to the coders to fix it and then overall software is retested.
- This is done to verify that the entire application works according to the customer requirement.

Phase - 6 Deployment :-

- After overall testing of the software and after checking that it is bug free, then the software is launched and available for the users to use it.
- Even after deployment of the software, if any bugs or errors are still found then the software is re evaluated by the maintenance team and then it is redeployed with a new version.

Phase - 7 Maintenance :-

- The maintenance team look over the software usage and user's feedback.
- Maintenance is necessary to eliminate errors in the system during its working life and to tune the software.

Software development process model :-

1. Waterfall Model :-

- The waterfall model is the simple and classical model of all the models we have.
- This model is also known as linear sequential model.
- This model is the theoretically model not a practical model.
- In this model each an every phase must be completed before moving to the next phase.
- This model is suitable for the small project where the technical issues are very clear.
- At the end of each phase a review will take place.

Advantages of Waterfall model

1. Waterfall model is the very simple and easy model which we can use for small project.
2. In this model once a phase is successfully completed then only we can move to the next phase so there is no overlapping between the phases.
3. Each phase has a well defined task and a review process.

Disadvantages of Waterfall model

1. It is not use for big projects.
2. We can't move back in last phase.
3. This model contains high risk.
4. It is poor model for complex and object oriented projects.

---

2. Prototype model :-

- It is a very famous software development model.
- In this model client is also involved in the time of designing the system.
- In this prototype model is modified until the client is not satisfied then we jumped to the next phase.
- The goal of prototype model to provide a system with overall functionality.
- Prototype model is an iterative development life Cycle between developer & client.

- We can also use the prototype model with other model.

Advantages :-

1. Users are not satisfied with the prototype, then a new prototype is created again.
2. Customer feedback is available for better system.
3. It takes less time to make this project because once we have done the requirement analysis from the customer, then it will take less time to develop the project.
4. Error can be detected at the earlier stage.

Disadvantages :-

1. Total dependency on prototype.
2. Sometimes it takes very long time to develop the prototype based on the user requirement.
3. The developer can try to reuse the prototype in another project even the project is not feasible.

---

3. Spiral model :-

- Spiral model was developed by “Barry Boehm” in the year 1986 as a part of SEI (S/W engineering institute).
- It is called meta model (model about model) because it contains all the life cycle model.
- The main purpose of Spiral model to reduce the risk in the project Spiral model has been introduced.
- One business analyst is required to reduce the risk with the help of developer and client then we can say how much cost it will take to developed.
- This model is mainly suitable for large and complete project.
- It is called an spiral because the same activities are repeated for all the loops (spiral).
- Each spiral or loop represents the software development process.

Advantages

- Risks are analyzed at the early stage of project development.
- Very famous model to develop large & complex projects.
- Best technology has been used inside the spiral model.
- Customer is available so we always get customer feedback at regular basis.

Disadvantages

- The cost of the project will be high because the System Analyst is required to analyse the risk.
- It is not suitable for small project.
- Each loop contains four quadrant so takes more time to complete.

Project Planning :- (Section - A)

Cocomo model :-

- Cocomo stands for "Constructive cost model". It is one of the very famous model which is used to estimate the cost of the project.
- Cocomo model was proposed by Boehm in the year 1981.
- Using this model we can estimate the time (month) and no of people (members) needed to develop a project.
- According to Boehm project cost estimation should be done through three stages :-

1. Basic Cocomo
2. Intermediate Cocomo
3. Complete Cocomo

---

1. Basic Cocomo :-

- The basic cocomo model gives an approximate estimate of the project parameters. it means it predict the effort and cost of project.

The following formula is used to estimate the cost in this model :-

```
effort = a1 x (Kilo line of code)^2  per month
Tdev = b1 x (effort) b2 months
```

---

2. Intermediate Cocomo :-

- This is an extension of the basic Cocomo model. The intermediate cocomo model uses 15 additional predictors considering its environment to estimate a value or cost.

---

3. Complete Cocomo :-

- The complete cocomo model is an extension of the intermediate cocomo model. This model is phase sensitive. It does not depend on any phase, it is used to calculate the amount of effort required to complete each phase.

Definition of Project Planning :-

- Project Planning is the process of defining clear activities, estimating time, effort, cost, and allocating resources required to complete a software project successfully.

It helps the project manager to :

1. Track project progress
2. Control cost and schedule
3. Ensure quality delivery
4. Manage risks effectively

- Project planning tools such as PERT and CPM are used to plan, schedule and monitor projects.
- The main goal of software project planning is to document estimates so that planning, tracking and commitments are clearly understood by all stakeholders.

---

4. Time, effort and resource estimation

- It determines the amount of time, effort, and resources required to complete each project activity.
- Estimation techniques such as function points, lines of code and benchmarks are used to prepare realistic project plans.

---

5. Risk factor identification

- It involves identifying potential risk related to assumptions, constraints, user expectations and project environment.
- Early identification of risks helps in minimizing their impact on the project.

---

6. Schedule development :-

- It is the process of creating a project schedule based on defined activities, their dependencies and estimated effort.
- Tools such as Gantt charts and PERT charts are used to plan and monitor project timelines.

---

7. Cost estimation and budgeting :-

- It involves estimating the total cost required to execute the project based on previous planning activities and allocating the budget accordingly to ensure effective financial control.

---

8. Risk management planning :-

- It includes analyzing identified risks, preparing risk response plans, and continuously monitoring risks throughout the project to reduce their impact on project objectives.

Activities of Project Planning :-

Project planning consists of several important activities that determine the success of a software project.

---

1. Project Scope definition and Scope Planning :-

- Defines what the project will deliver and what it will not.
- Documents :

(i) Project objectives
(ii) Assumptions and constraints
(iii) Business and technical requirements
(iv) User Expectations

- Acts as the foundation for the entire project.

---

2. Quality planning :-

- It focuses on identifying the quality standards required for the software project and defining the processes needed to ensure that the final product meets customer expectations and agreed standards based on scope and requirements.

---

3. Project activity definition and activity sequencing :-

- Identifies all activities required to produce project deliverables.
- Determines the order and dependency of activities.
- Helps understand which tasks can run in parallel.

---

9. Project plan development and execution :-

- It integrates all planning outputs into a detailed project plan using tools such as work breakdown structure (WBS), schedules, and timelines which guides the execution and monitoring of the project.

Software Requirement
Analysis
Section-A

➤ SRS [Software Requirement Specification]

- SRS stands for software requirement specification. It is a document prepared by System analyst.
- It describes what will be the features of software and what will be its behaviour i.e. how it will perform.
- It is a detailed description of software system to be developed with its functional & non-functional requirement.
- The SRS consists of all necessary requirements required for the project development.
- In order to get all the details of software from customer and to write the SRS document system analyst is required.
- SRS document is actually an agreements b/w the client & developer.

➤ Characteristics of SRS :-

1. **Complete :-** The SRS document must be completed by taking all the requirement to related to software development.
2. **Consistent :-** It should be consistent from beginning to end, so that user can easily understand the requirement. And consistency can be achieved only when there is no conflict b/w two requirements.
3. **Feasible :-** All the requirements included in SRS document must be feasible to implement.
4. **Modifiable :-** The SRS document must be created in such a manner that it should be modifiable.
5. **Testable :-** The entire software or the individual module of the software must be testable.
6. **Correct :-** To all the requirements given in the SRS document should be correct, so we can easily implement in software part.
7. **Realistic :-** An SRS document should be reviewed to ensure that they are possible.
8. **Verifiable :-** All the documents of SRS must be verified.
9. **Unambiguous :-** The module of the software must be unambiguous so one module is compatible with another module.

➤ Components of an SRS :-

- Business Drivers
- Business models
- Constraints & assumptions
- Business and System use cases
- Business / Functional and software requirements
- Technical requirements
- System qualities

- Acceptance criteria

1. **Business Drivers :-** This section describes the reason the customer is looking to build the system including problems with the current system and opportunities the new software will provide.
2. **Business model :-** This section describe the business model of the customer and the system has the support including organizational, business context, main business functions and process flow diagrams.
3. **Business / Functional And System Requirements :-** This section typically consists of requirements that are organized in a hierarchical structure.
4. **Business and System use Cases :-** This section consists of a unified modeling language (UML) use case diagram depicting the key external entities that will be interacting with the software and the different use cases that they will have to perform.
5. **Technical Requirements :-** This section lists the non functional requirements that make up the technical environment where software needs to operate and the technical restrictions under which it needs to operate.
6. **System qualities :-** This section is used to describe the non-functional requirements that define the quality attributes of the system such as reliability, serviceability, security, scalability, availability and maintainability.
7. **Constraints and assumptions :-** This section includes any constraints the customer has imposed on the system design. It also includes the requirements engineering team's assumptions about what is expected to happen during the project.
8. **Acceptance Criteria :-** This section details the conditions that must be met for the customer to accept the final system.

➤ **DFD [Data flow Diagram] :-**

- DFD stands for data flow diagram. It is also known as Bubble chart through which we can represent the flow of data graphically in an information system.
- By using DFD we can easily understand the overall functionality of system because diagrams represents the incoming data flow, outgoing data flow and stored data in a graphical form.
- It tells us how data is processed ing system in terms of input & output.
- A DFD model uses a no. of notations or symbol to represent flow of data :-

1. **External Entity** □
2. **Data Flow** →
3. **Process or Bubble** ○
4. **Data Store** ═ or □□

➤ **Rules of DFD :-**

- Each process should have atleast one input and one output.

  Ex → Input → (Process) → Output

- Each data store should have atleast one dataflow in and one data flow out.

  Ex → In → ═ → Out

- All process in a DFD go to either another process or data store.

  Ex → (Process 1) → (Process 2)
  ↘
  ═══ data store

- All the external entities must be connected through a process and entity can provide something to the software as well as the entity can consume some data from the software.

Levels of DFD :-

1) 0th level dfd :-
• It is a diagram which provides the entire system's data flows and processing with a single process (bubble is) called as context.

Ex:-

---

2) 1st level dfd :-
• This is a more detailed version of the previous level that includes the database and various important units.

Ex:-

Data Dictionary :-

• A data dictionary contains metadata i.e data about the database.

• It contains information such as what is in the database, who is allowed to access it, where is the database physically stored etc.

• It plays an important role in building a database.

Example table:

Client-id	Client-name	Password	Contact-no	Email
1	Raju	1234	77781	rj@gmail.com
2	Rohan	5678	77992	ro@gmail.com

Field Description:

Field Name	Datatype	Length	Constraints	Description
Client-id	number	10	Primary Key	Client id auto generated
Client-name	varchar	20	not null	Client Name

Field Name	Datatype	Length	Constraints	Description
Password	varchar	30	not null	Login Pswrd
Contact no	number	10	not null	Contact of client
Email	varchar	40	not null	Client email

• The data dictionary in general contains information about the following:

1. Names of all tables in the database.
2. Names of each field in the tables of the database.
3. Constraints defined on tables.
4. Physical information about tables like their storage location, storage method etc.

• The users of the database normally do not interact with the data dictionary, it is only handled by the database administrator. A data dictionary also called a metadata repository.

Types :-

1) Active :-
• It may happen when the structure of the database has to be changed like adding new attributes or removing older ones.

• If those changes are updated automatically in the data dictionary by the DBMS then the data dictionary is an active one.

• It is also known as integrated data dictionary.

2) Passive :-
• When the DBMS maintains the data dictionary separately and it has to be updated manually, then the data dictionary is a passive one.

• It is also known as non integrated data dictionary.

• In this case, there is a chance of mismatch with the database objects and the data dictionary.

• It gives the well structured and the clear information about the database.

ERD :-

• ER Model stands for entity relationship model.

• It is a graphical approach for designing a database.

• It is a database modelling technique that generates an abstract diagram of system’s data and that diagram is used in designing a relational database.

• It was developed by Peter Chen in 1976.

• ERD stands for Entity relationship diagram.

• The diagram generated by ER model is known as ERD.

• ERD shows the relationship of different entities stored in the database.

Example: Customer buys Item.

Customer table:
Ram – 12/ABC
Sham – 15/XYZ

Item table:
Mouse – 125
Keyboard – 250

• ER diagram explains the logical structure of database.

• At last, an ERD is implemented as a database.

• It helps us to analyze data requirements systematically to produce a well designed database.

• It is considered as a best practice to complete ER modeling before implementing database.

• ER diagrams are based on three basic concepts: entities, attributes and relationships.

1) Entity :-

• An entity is a real world object or concept about which you want to store information.

• An entity can be place, person, object or a concept etc.

Example:
Person – student, employee, patient
Place – building, store
Object – car, bike
Concept – course

• It is represented using rectangle shape.

---

2) Attributes :-

• Attributes are the properties and characteristics of an entity.
• It describes an entity. It is represented using oval shape.

Example: Student → Name, Address, Roll no.

3) Relationship :-

• It specifies how entities are associated with each other.
• It is represented using diamond shape.

Example: Student enroll Course.

Keys :-

1) Primary Key :-

Primary key is the key field which uniquely identify each record of the table.

Example Student table:

Roll no	Name	Course
101	Ram	BCA
102	Sham	BBA

---

2) Candidate Key :-

There may be two or more attributes or combination of attributes that uniquely identify an instance of an entity set. These combinations of attributes are called candidate keys.

---

3) Alternative Key :-

A candidate key which is not the primary key is called the alternative key.

---

4) Foreign Key :-

Foreign key is the field which uniquely identify the records on another table.

---

5) Unique Key :-

Unique key is the key fields which are also used to identify unique records from a table.

---

6) Secondary Key :-

The key which is not giving the unique identification and have duplicate information is called secondary key.

Need for SRS :-

• The software requirement specification is required because it acts as the parent document for the entire software development process.

• All subsequent documents such as design specifications, software architecture documents, testing and validation plans are derived from the SRS.

• An SRS provides clear feedback to the customer by assuring that the development team has correctly understood the customer's problems, needs, and expected software behaviour.

• It is written in natural languages and may include diagrams, tables, charts, data flow diagrams, decision tables to avoid ambiguity.

• A well defined SRS enables accurate estimation of cost, effort, and schedule. Techniques such as function point analysis can be applied to the SRS to estimate development effort and project pricing.

• The SRS serves as a formal contract between the customer and the software vendor. It specifies all functional and non functional requirements, technical constraints and interface requirements, helping both parties verify whether the delivered software meets the agreed specifications.

• An SRS helps in decomposing the problem into manageable components.

• Writing requirements in a structured and well defined format organizes information, sets boundaries around the problem and simplifies complex system into smaller parts.

Metrics :-

• Metrics are quantitative measures used to obtain numerical information about different attributes of software products, processes or documents.

• In the requirements phase metrics helps in assessing the size, quality and effort required for a project.

• Although there is limited scope for metrics during the requirement phase, some useful metrics can still be applied to estimate cost, effort, and project size once the requirements are finalized.

• One important metric related to SRS is size estimation, which uses the size of requirements as a base to estimate project cost. The major size metrics used are text based metrics and function points.

• Text based metrics measures the size of the SRS based on the number of pages, paragraphs, lines or requirements written in the document. Since SRS is written in natural languages, the metrics provides a simple way to estimate project size but it may be subjective.

• Function point metrics measure size based on the number of functionalities described in the requirements.

• Function points are considered a better metric because they reduce subjectivity related to individual writing styles and focus on system functionality.

Validation :-

• Validation is the process of checking whether the software requirements specified in the SRS document correctly reflect the actual needs of the user and intended use of the system.

• It ensures that the right system is being built before moving to design and implementing phases.

• The SRS document acts as a guide for all subsequent phases of software development, so errors introduced at the requirement stage can propagate to later phases and become expensive to fix.

• Therefore, validation is essential to detect errors early in the requirement phase itself.

• Requirement validation helps identify ambiguous, inconsistent, incomplete or incorrect requirements that may differ from the client's real expectations.

• Without proper validation, the final system may fail to satisfy user needs even if it is correctly implemented.

• During requirement validation, the following aspects are checked:

- Whether some user requirements are missing.
- Whether requirements contradict each other.
- Whether incorrect facts are recorded.
- Whether requirements contain ambiguity.

Software Process
Section – A

• A software process is a structured set of activities that are followed to develop, operate, maintain and improve software systems.

• It defines what work is done, who does it, how it is done, and in what order.

• Software process is a set of activities whose goal is the development or evolution of software.

• It includes:
(i) Software programs and procedures
(ii) Documentation such as manuals and flowcharts
(iii) Human resources (developers, testers, managers)
(iv) Tools, techniques and constraints

Characteristics of Software Process :

1. Optimality :-
The process should produce high quality software at minimum cost and effort.

2. Maintainability :-
The process should allow easy modification and improvement when changes are required/requested.

3. Understandability :-
The process should be clearly defined and easy to understand by developers and managers.

4. Visibility :-
Each activity should produce clear outputs, so project progress can be monitored.

5. Supportability :-
The process should be supported by automated tools and CASE tools.

6. Robustness :-
The process should continue to function even when unexpected errors occur.

7. Rapidity :-
The process should allow fast development and delivery of software.

8. Reliability :-
The process should help reduce errors and produce reliable software.

9. Acceptability :-
The process should be acceptable to software engineers and easy to follow.

10. Testability :-
The process should support testing against predefined criteria.

Project Management Process :

• The project management process is a systematic approach used to plan, organize, direct, staff and control a software project so that it is completed within time, budget and according to required quality standards.

• It ensures proper utilization of resources, manpower, tools and techniques to achieve project objectives.

(Project management cycle: Planning → Organizing → Staffing → Directing → Controlling)

---

1. Planning :-

Planning is the first and most important activity of the project management process.

• It involves deciding what objectives are to be achieved, what resources are required, how the objectives will be achieved, and when the work will be completed.

Planning also includes defining:
• Project scope
• Cost estimation
• Risk management
• Time schedules

Importance
Without proper planning, a project may suffer from delays, cost overruns and failure.

Example
Before developing an E-commerce website, the project manager plans:
• No. of developers and testers required
• Development time
• Budget allocation
• Tools and technologies to be used

---

2. Organizing :-

• Organizing involves the establishment of clear authority, responsibility and relationships among project team members.

• It defines who will do what and who will report to whom.

• Organizing ensures proper coordination among different project activities.

Importance
Good organization ensures proper coordination among different activities and avoids confusion, duplication of work and conflicts within a team.

Example
In a software company:
• One person is assigned as project manager
• Developers handle coding
• Testers handle testing
• UI designers handle interface design

Each member clearly knows their role.

---

3. Staffing :-

• Staffing deals with recruiting, selecting, training, compensating and managing personnel required for the project.

• It ensures that the right people with the right skills are assigned to the project.

Importance
A project's success heavily depends on the skills and efficiency of its team members.

Example
If a mobile app project requires a good user interface:
• A skilled UI/UX designer is hired
• Developers are trained on new frameworks
• Team performance is monitored

---

4. Directing :-

• Directing involves leading, guiding, motivating and supervising team members so that project goals are achieved.

• It ensures that team members work in the right direction.

Importance
Effective directing improves team morale, productivity and coordination.

Example
During development:
• The project manager conducts meetings
• Provides technical guidance
• Resolves conflicts
• Motivates the team to meet deadlines

5. Controlling :-

• Controlling involves measuring actual performance, comparing it with planned performance and taking corrective actions when deviations occur.

Importance
It ensures the project remains on track and meets objectives.

Example
If testing is delayed:
• The manager identifies the cause
• Allocates more resources
• Adjusts the schedule to avoid final delivery delay

Software Configuration Management Process :

• Software configuration management process is a discipline that focuses on identifying, organizing, controlling and tracking changes made to software products throughout their life cycle.

• SCM ensures that changes are made systematically and safely without affecting software quality.

Importance of SCM

• Controls frequent software changes
• Maintains consistency and integrity
• Prevents loss of data and confusion

SCM Activities

1. Configuration Identification

• The activity identifies all software items that need to be controlled such as:
• Source code
• Design documents
• Test cases
• User manuals

Each item is uniquely labeled and versioned.

Example
Naming files as:
• Login-V1.java
• Login-V2.java

---

2. Change Control

• Change control ensures that all changes are formally requested, reviewed, approved and implemented.

• Unauthorized changes are strictly avoided.

Example
A client requests a new feature:
• Change request is submitted
• Manager evaluates impact
• Change is approved or rejected

---

3. Version Control

• Version control manages multiple versions of software and keeps track of changes made by different developers.

Example
Using Git to maintain different branches like:
• Main branch
• Feature branch
• Bug fix branch

---

4. Configuration Status Accounting

• This activity records and reports the current status of configuration items and changes.

Example Tracking:
(i) Which features are completed
(ii) Which changes are pending

---

5. Configuration Auditing

• Auditing verifies whether approved changes are correctly implemented and documented.

Example
Before releasing software:
• Checking code consistency
• Verifying documentation
• Ensuring all approved changes are included

Software Metrics :

• A metric is a measurement of the level at which any input belongs to a system product or process.

• It is a quantitative measure of the degree to which a system, component or process possesses a given attribute.

• Software metrics in software engineering are the standards for estimating the quality, progress, and health of software development activity.

• A software metric is a quantifiable or countable assessment of the qualities of software.

Categories of Software Metrics :

1. Product metrics :-
• Product metrics are used to evaluate the state of the product, tracking risk and uncover prospective problem areas.

• The ability of the team to control quality is evaluated.

• Examples include lines of code, cyclomatic complexity, code coverage, defect density, and code maintainability index.

---

2. Process metrics :-
• Process metrics pay particular attention to enhancing the long term process of the team or organization.

• These metrics are used to optimize the development process and maintenance activity of software.

• Examples include effort variance, schedule variance, defect injection rate, and lead time.

Software Quality :-

• Software quality shows how good and reliable a product is.

• Software quality can be defined as:

→ Degree of excellence
→ Fitness for purpose
→ Best for the customer’s use and selling price
→ The totality of characteristics of an entity that bears on its ability to satisfy stated or implied needs.

Software Quality Attributes include:
Usability, Functionality, Reliability, Maintainability, Efficiency, Portability.

Quality may be defined from different perspectives. The quality of the product contributes to improved user requirements and satisfaction, cleaner software design, and ultimately greater end product quality.

Attributes :-

1. Reliability :-

• Software is more reliable if it has fewer failures.

• Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and types of mistakes they make.

• Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or recover easily.

---

2. Maintainability :-

• A software is repairable if errors may be simply corrected as and once they show up, new functions may be simply added to the merchandise, and therefore the functionalities of the merchandise may be simply changed etc.

---

3. Usability :-

• This attribute refers to the quality of the end user's experience while interacting with the application or service.

• Usability is concerned with effectiveness, efficiency, and overall user satisfaction.

• This attribute helps to measure the ease of use of any software application or service.

---

4. Portability :-

• It refers to the extent to which a system or its components can be migrated (transported) to other environments consisting of different hardware or different software (operating system).

• Sub attributes for portability are adaptability, replaceability, installability, and compatibility.

---

5. Correctness :-

• This refers to the ability of software products or services to perform tasks (e.g., calculations, sign-ups, navigations) correctly as specified by the predefined requirements when used under specified conditions.

• It is a measure of whether a software is correct or not.

---

6. Efficiency :-

• The more efficient software is the less it uses CPU time, memory, disk space, network bandwidth and other resources.

• This is important to customers in order to reduce their costs of running the software although with today's powerful computers, CPU time, memory and disk usage are less of a concern than in years gone by.

• Sub attributes for efficiency are time, behaviour, resource utilization, capacity etc.

---

7. Security :-

• It consists of authorization and authentication techniques, safeguarding against network attacks, sending encrypted data wherever necessary, and preventing SQL injection.

Cyclomatic Complexity Metric :-

• Cyclomatic complexity is a software metric used to determine the complexity of a program.

• Cyclomatic complexity is a count of the number of decisions in the source code. The higher the count the more complex the code.

• Thomas J. McCabe developed this metric in 1976. McCabe interprets a computer program as a set of a strongly connected directed graph.

• Nodes represent parts of the source code having no branches and arcs represent possible control flow transfers during program execution.

• The notation of program graph has been used for this measure and it is used to measure and control the number of paths through a program.

Cyclomatic Complexity can be used in two ways such as:

• Limit code complexity
• Determine the number of test cases required

How to calculate Cyclomatic Complexity :-

McCabe proposed the cyclomatic number V(G) of graph theory as an indicator of software complexity.

The cyclomatic number is equal to the number of linearly independent paths through a program in its graph representation.

For a program control graph G, cyclomatic no., V(G) is given as:

V(G) = E – N + 2 * P

Where:
E = The number of edges in graph G
N = The number of nodes in graph G
P = The number of connected components in graph G

Example cases:

Sequence:
V(G) = 1 – 2 + 2 = 1

While loop:
V(G) = 3 – 3 + 2 = 2

Until loop:
V(G) = 3 – 3 + 2 = 2

Software Maintenance :-
Section – B

• Software maintenance refers to the modification of a software product after it has been delivered to the customer in order to correct faults, improve performance or other attributes, or adapt the software to changes in the environment.

• According to IEEE, software maintenance is the modification of a software product after delivery to correct faults, enhance performance, or adapt it to a changed environment.

• Software maintenance includes the total set of activities required to provide cost effective support to a software system throughout its life cycle.

• These activities are performed during both predelivery and post delivery stages, where predelivery activities include planning for supportability, logistics and postdelivery operations.

Categories of Software Maintenance :-

1. Corrective Maintenance :-

• Corrective maintenance deals with the correction of faults or defects discovered in the software after it has been delivered and put into operation.

• These faults may arise due to design error, logic error, coding error and data processing error.

• This type of maintenance is usually initiated by bug reports from users and focuses on restoring the software to its specified working conditions.

• Corrective maintenance does not add new functionality but only fixes errors that prevent correct operation.

---

2. Adaptive Maintenance :-

• Adaptive maintenance involves modifying the software to keep it usable in a changing environment.

• The environment may change due to new hardware, operating systems, business rules, government policies or software platforms.

• This type of maintenance is necessary when external conditions affect the software system, and it ensures that the software continues to function correctly without changing its core functionalities.

• Examples include adapting software to a new operating system, integrating a new database management system, or modifying software to comply with new government regulations.

---

3. Perfective Maintenance :-

• Perfective maintenance focuses on enhancing the software by improving performance, usability or adding new features based on changing user requirements.

• As users gain experience with the system, they often request additional functionality or improvements.

• This type of maintenance increases the value and usefulness of the software and is usually performed during the middle and later stages of the software's operational life.

• Examples include improving user interfaces, optimizing performance, adding new reports, or extending existing functionalities.

• These studies are especially useful in critical applications such as industrial systems, military software, and commercial systems, where software failure can have serious consequences.

• Reliability analysis helps in continuous improvement of software performance, as data collected during testing and operation is used to refine development processes and improve future software products.

• They help software managers in decision making such as determining readiness, maintenance planning, and allocation of testing resources based on reliability data and predicted failure rates.

---

4. Preventive Maintenance :-

• Preventive maintenance includes making changes to improve the software’s maintainability and prevent future problems.

• It focuses on reducing system complexity and improving code quality to avoid potential failures.

• This type of maintenance is initiated by the maintenance team itself and helps make future maintenance tasks easier, more efficient and less costly.

• Preventive maintenance includes activities such as code restructuring, optimization, adding comments, improving documentation, and enhancing modularity.

Coding
Section – B

• Coding is the phase of the software development process in which the design and algorithms are translated into executable computer programs using programming language.

• It is the activity where the software design becomes a working product.

• Coding requires careful attention because errors introduced at this stage can be costly and difficult to fix later.

• Good coding improves software quality, maintainability, reliability and performance.

• Coding is the process of writing program instructions in a programming language based on software design specifications so that the computer can execute the desired tasks.

Programming Guidelines (During Coding) :

1) Make a Plan

• Before writing any code, the programmer must prepare a clear plan of how the problem will be solved.

• The planning stage involves understanding the problem, deciding the approach, and identifying the major operations required.

Planning helps in:
• Reducing unnecessary code
• Saving memory and processing time
• Achieving maximum accuracy

Example:
Before coding a payroll system, the programmer plans how salary, tax, bonuses and deductions will be calculated and stored.

---

2) Break the Problem Down

• Large and complex problems should be divided into smaller, manageable sub problems, also known as modules or subroutines. This approach is called modular programming.

Breaking down a problem:

1. Makes the program easier to understand
2. Simplifies testing and debugging
3. Encourages reuse of code

Example:
An online shopping system is divided into modules such as user login, product catalog, payment processing and order tracking.

---

3) Keep the Code Simple

• Simplicity should always be a key goal in coding. Simple code:
• Takes less time to write
• Contains fewer bugs
• Is easier to modify and maintain

Complex and unnecessary logic should be avoided.

---

4) Program the Subroutines First

• Subroutines should be coded before the main routine because many implementation details become clear only when subroutines are developed.

• Changes in subroutines may affect the overall program structure.

Writing subroutines first:
• Reduces errors
• Helps in better integration
• Makes the main program simpler

Programming Principles :

1) Single Responsibility Principle

• This principle states that each class or function should have only one responsibility.

• When a component has multiple responsibilities, changes become difficult and error prone.

Example:
A class that handles user authentication should not also handle database operations.

---

2) Do the Simplest Thing that Could Possibly Work

• The simplest solution that satisfies the requirement should always be preferred. Complex solutions increase chances of bugs.

---

3) Don’t Make Me Think

• Code should be self explanatory and easy to read. The reader should not struggle to understand what the code does.

---

4) Open / Closed Principle

• Software components should be open for extension but closed for modification. This allows new functionality to be added without altering existing code.

---

5) Principle of Abstraction

• Abstraction hides implementation details and exposes only necessary functionality. It improves code reuse and maintainability.

Verification – Code Inspection

• Verification ensures that software is built correctly according to specifications and standards.

It answers the question:
Are we building the product right?

• Verification is the process of evaluating software work products to ensure that they meet specified requirements and standards.

• Code inspection is a static verification technique in which code is examined without executing it to detect errors early in the development process.

Fagan Inspection (Formal Inspection)

Fagan inspection is a structured and systematic method of code inspection that follows a predefined process.

Stages of Fagan Inspection

1. Planning
   In this stage:
   • Inspection team is selected
   • Documents are prepared
   • Schedule is finalized
2. Overview
   • The author explains the product to inspectors. Roles such as moderator, reader, and recorder are assigned.
3. Preparation
   • Each inspector individually studies the code to identify potential defects.
4. Inspection Meeting
   • Defects are identified, discussed and documented. No solutions are suggested at this stage.
5. Rework
   • The author corrects the defects identified during inspection.
6. Follow-up
   • The moderator verifies that all defects are fixed and no new defects are introduced.

Static Analysis

• Static analysis is the process of analyzing source code without executing the program.

• It is performed to detect errors, anomalies, and vulnerabilities early.

• Static analysis is the examination of software source code to detect defects and inconsistencies without running the program.

Purpose of Static Analysis

1. Detect programming errors early
2. Improve code quality
3. Identify security vulnerabilities
4. Reduce testing and maintenance cost

What Static Analysis Tools Check

• Syntax correctness
• Control flow errors
• Data usage problems
• Unused variables
• Unreachable code

Stages of Static Analysis

1) Control Flow Analysis

This analysis examines the order in which program statements execute. It identifies:
• Infinite loops
• Unreachable code
• Multiple entry or exit points

---

2) Data Use Analysis

This analysis checks how variables are used. It detects:
• Variables used without initialization
• Variables declared but never used
• Variables written but never used

---

3) Interface Analysis

Interface analysis checks consistency between:
• Function declarations and calls
• Parameter types and return values

---

4) Information Flow Analysis

This analysis tracks how information flows between variables. It is useful for detecting security issues and data leakage.

---

5) Path Analysis

Path analysis examines all possible execution paths in a program to detect logical inconsistencies.

Software Testing :-
Section – B

Software Testing is the process of evaluating the software product for identifying bugs/issues in the product.

Testing is the process of finding errors/faults in a software application with the aim to deliver a quality product to the customer.

Testing makes software predictable in nature.

Testing is the process of verifying a system with the purpose of identifying any errors, gaps or missing requirements versus the actual requirement.

In a traditional SDLC, testing is normally conducted at the end of the software development phase.

Black Box Testing :-

Black box testing is a type of software testing in which the tester is not concerned with the internal knowledge or implementation details of the software but rather focuses on validating the functionality based on the provided specifications or requirements.

Black box testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications.

It is also known as behavioral testing.

Input → Black box Testing → Output

Black box can be any software system you want to test. For example, an operating system like Windows, a website like Google, a database like Oracle or even your own custom application.

Under black box testing, you can test these applications by just focusing on the inputs and outputs without knowing their internal code implementation.

Black Box Testing Techniques :-

1. Equivalence Class Testing :-

It is used to minimize the number of possible test cases to an optimum level while maintaining reasonable test coverage.

2. Boundary Value Testing :-

Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not.

It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.

Decision Table Testing :-

A decision table inputs causes and their effects in a matrix. There is a unique combination in each column.

Types of Black Box Testing :-

There are many types of black box testing but the given are the prominent ones:

1. Functional Testing

This black box testing type is related to the functional requirements of a system. It is done by software testers.

2. Non Functional Testing

This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.

3. Regression Testing

Regression testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

White Box Testing :-

White box testing or structural, code based, or glass box testing is a software testing technique that focuses on the software's internal logic, structure and coding.

It provides testers with complete application knowledge.

Including access to source code and design documents, enabling them to inspect and verify the software's inner workings, logic structure and integrations.

(Test Case Input → Program Structure → Test Case Output)

Process of White Box Testing :-

1. **Input** → Requirements, functional specifications, design documents, source code.

2. **Processing** → Performing risk analysis to guide through the entire process.
3. **Proper test planning** → Designing test cases to cover the entire code. Execute tests – repeat until error free software is reached. Also, the results are communicated.
4. **Output** → Performing final report of the entire testing process.

Control Flow Testing :-

Control flow testing is a White Box testing technique in which the internal structure of the program is examined to ensure that all possible paths of execution are tested.

It focuses on the flow of control through the program such as sequence, decision and loop statements.

The objective of control flow testing is to verify that every statement, branch and path in the program is executed at least once, thereby detecting logical errors in decision making constructs.

In control flow testing the program is represented using a control flow graph:

• Nodes represent program statements
• Edges represent flow of control

By analyzing this graph, testers design test cases to cover:

• All statements
• All branches (true/false conditions)
• All independent paths

Example:
Consider a program that checks whether a number is positive or negative using if else statement. Control flow testing ensures that:

• The condition is tested when it is true
• The condition is tested when it is false

This helps detect missing conditions or unreachable code.

Data Flow Testing :-

Data flow testing is a White Box testing technique that focuses on the lifecycle of variables, that is how data values are defined, used and modified within a program.

It checks whether variables are:

• Properly initialized before use
• Used after assignment
• Not declared but never used

Data flow testing examines definition–use (DU) relationships of variables:

1. **Definition (D):** When variable is assigned a value.
2. **Use (U):** When a variable is used in a computation or condition.

Common data flow anomalies detected:

1. Variables used without initialization
2. Variables defined but never used
3. Variables defined multiple times without use

Example:
If a variable total is used in a calculation before being assigned a value, data flow testing will detect this error.

Mutation Testing :-

Mutation testing is a fault based testing technique in which small changes are deliberately introduced into the program to check whether the existing test cases can detect these faults.

The goal is to evaluate the effectiveness of test cases, not just the correctness of the program.

In mutation testing:

1. The original program is called the base program.
2. Modified versions are called mutants.
3. If a test case detects a mutant, the mutant is said to be killed.
4. If a mutant produces the same output as the original program, it is a surviving mutant.

Example:
Changing > to < in a condition:

if (a > b)
to
if (a < b)

If test cases fail to detect this change, they are weak.

Levels of Testing :-

1. Unit Testing :-

Unit testing focuses on testing individual components or modules of software in isolation.

• Usually performed by developers
• Ensures correctness of each unit

Example: Testing a single function that calculates interest.

---

2. Integration Testing :-

Integration testing tests the interaction between combined modules to detect interface defects.

• It ensures that data flows correctly between modules.

Example: Testing interaction between login module and database module.

---

3. System Testing :-

System testing tests the entire integrated system as a whole against functional and non-functional requirements.

• Performed by testers
• Tests performance, security, reliability

Example: Testing an entire banking application.

---

4. Acceptance Testing :-

Acceptance testing is performed by the customer or end user to ensure software meets business requirements.

Types include:
• Alpha Testing
• Beta Testing

Example: Customer testing software before final approval.

Test Case Specification :-

A test case specification is a detailed document that describes test conditions, input data, execution steps, and expected results required to verify a specific feature of the software.

Explanation

A well-written test case ensures:

1. Consistent testing
2. Repeatability
3. Clear validation criteria

Components of Test Case

A test case typically includes:

1. Test case ID
2. Test description
3. Input data
4. Pre-conditions
5. Expected output
6. Actual output
7. Test status (Pass/Fail)

Example:
Test Case: Login Validation
• Input: Valid username and password
• Expected result: User successfully logged in

Test Case Execution :-

Test case execution is the process of running test cases on the software and comparing the actual results with expected results.

During execution:

1. Test cases are executed manually or automatically
2. Results are recorded
3. Defects are logged

Test execution helps determine whether the software behaves as expected under given conditions.

Example:
Executing a test case for payment processing and verifying whether payment is successful.

Test Case Analysis :-

Test case analysis is the activity of evaluating test execution results to identify defects, assess software quality, and improve test cases.

It involves:

1. Analyzing failed test cases
2. Identifying root cause of defects
3. Refining test cases
4. Measuring test effectiveness

Example:
If multiple test cases fail due to incorrect validation logic, analysis helps identify design flaws.

Test Plan :-

A test plan is a formal and detailed document that describes the scope, approach, resources, schedules, and activities involved in testing a software product.

- It acts as a blueprint for the entire testing process and defines what to test, when to test and who will test.
- The test plan is prepared before testing begins and serves as a reference document for testers, developers, managers, and clients.
- A test plan is a document that outlines the objectives, scope, strategy, resources, schedules, and responsibilities of software testing activities.
- The main purpose of a test plan is to:
    - Ensure systematic and planned testing.
    - Avoid confusion and duplication effort.
    - Provide clear direction to the testing team.
    - Act as a communication medium between stakeholders.
- A well defined test plan helps to deliver high quality software within time and budget constraints.

Contents of a Test Plan :-

1. Test Plan Identifier :-

This section uniquely identifies the test plan documents using:

- Project name
- Version number
- Date

It helps in version control and traceability.

---

2. Introduction

The introduction provides a brief overview of:

- The software product to be tested
- The purpose of the test plan
- Intended audience

It gives readers a clear understanding of why testing is required.

---

3. Test Items

Test items specify what components of the software will be tested, such as:

- Modules
- Features
- Interfaces
- Documents

This section ensures clarity on testing scope.

---

4. Features to be Tested

This section lists functional and non-functional features that will be tested, such as:

- User Authentication
- Input Validation
- Performance
- Security

Ensures that all critical functionalities are covered.

Test Approach / Test Strategy :-

The test approach describes how testing will be carried out. It includes:

- Types of testing (unit, integration, system, acceptance)
- Testing techniques (black box, white box)
- Manual or automated testing

Software Design :-
Section – B

- Software design is a very important phase in the SDLC. It is a technique through which we can design a meaningful representation of something that we want to built.
- It is a process by which the requirement (SRS) are translated into blueprint for creating a software.
- The blueprint gives us the complete details of working software.
- The design must meet all the user requirement.
- It also reduces effort & errors that come during the design.
- The entire software must be breakdown into no of modules known as top level design.
- The required algorithm and data structure is used to implement a particular module known as detail design.
- The entire software designing concept is divided into two parts:

1. Top or High level design
2. Detail design or internal design

⇒ Design Principles :-

- Design principles are general guidelines that help software designers create systems that are easy to understand, modify, test and maintain. Poor design leads to software that is rigid, fragile, and difficult to enhance.
- According to software engineering, four major symptoms of bad design are:

• Rigidity
• Fragility
• Immobility
• Viscosity

These help us judge whether a design is good or poor.

---

1. Rigidity :-

- Rigidity is the tendency of software to be difficult to change even for small modifications.
- A small change in one module requires changes in many other modules.
- This happens due to tight coupling between modules.
- Developers hesitate to make changes because of the fear of breaking the system.

Example
If changing the tax rate in a billing system requires modifying billing, reporting, invoice, and database modules, the system is rigid.

---

2. Fragility :-

- Fragility is the tendency of software to break in many places when a change is made.
- Errors appear in parts of the system that are unrelated to the change.
- Testing becomes difficult and time consuming.
- Managers and users lose confidence in the software.

Example
Fixing a login bug causes the payment module to crash, even though both are unrelated.

---

3. Immobility :-

- Immobility means the software cannot be reused easily in another project.
- Useful modules are tightly dependent on other parts.
- Extracting reusable components becomes expensive and risky.
- Developers prefer rewriting instead of reusing.

Example
A well written report-generation module cannot be reused because it is tightly linked to a specific database structure.

---

4. Viscosity :-

- Viscosity refers to how easy or hard it is to do the right thing in software design.

Types

1. **Viscosity of Design** – Good design changes are harder than bad shortcuts.

---

2. Viscosity of Environment :-

Tools or processes make correct changes slow.

Example
If proper refactoring takes hours but quick hacks take minutes, developers choose hacks.

⇒ Module Level Concepts :-

- A module is a self contained unit of software that performs a specific function.
- A module is a well defined, manageable, and independent component of a software system that interacts with other modules through well defined interfaces.

---

1. Coupling :-

- Coupling is the degree of interdependence between modules.
- Looser coupling is desirable because:
    - Changes in one module do not affect others.
    - System becomes flexible and maintainable.

Types of Coupling

1. Data Coupling

- Modules communicate by passing only required data.
- Best form of coupling.

Example – Passing customer ID to a function.

---

2. Stamp Coupling

- Modules share composite data structure.
- Dependency increases.

Example – Passing entire customer object when only ID is needed.

---

3. Control Coupling

- One module controls the behavior of another using flags.

Example – Passing a flag to decide which operation to perform.

---

4. Common Coupling

- Two modules are common coupled when they share some global data items.

Example – Multiple modules accessing the same global counter.

---

5. Content Coupling

- One module modifies internal data of another.
- Worst type of coupling.

Example – Directly changing another module's local variables.

---

2. Cohesion :-

- Cohesion measures how closely related the elements within a module are.
- Higher cohesion is desirable.

Types of Cohesion

1. Functional Cohesion

- When instructions in different modules are related because they collectively work together to accomplish a single well-defined function.
- All elements perform a single well defined task.

---

2. Sequential Cohesion

- A module is said to sequential cohesion when all the elements of a module form the parts of a sequence, where the output from one element of the sequence is input to next.

---

3. Communicational Cohesion

- When instructions in different modules accomplish tasks that utilize the same piece of data.

---

4. Procedural Cohesion

- When instructions accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

---

5. Temporal Cohesion

- When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to be temporal cohesion.

---

6. Logical Cohesion

- When a module contains instructions that appear to be related because they fall into the same logical class of functions.

---

7. Coincidental Cohesion

- When a module contains instructions that have little or no relationship to one another.

⇒ Structure Chart :-

- Structure chart is a hierarchical diagram that shows how a software system is divided into modules and how these modules interact.
- A structure chart depicts:
    1. The size and complexity of the system
    2. Number of readily identifiable functions and modules within each function
    3. Whether each identifiable function is a manageable entity.
- A structure chart is also used to diagram associated elements that comprise a run stream or thread.

Example

A payroll system structure chart may have:

- Main module
- Input module
- Calculation module
- Output module

Each sub module perform a specific task.

⇒ Structured Design Methodology :-

- Structured design is a design technique based on functional decomposition, where the system is broken down into smaller modules in a top-down manner.

Steps in Structured Design

1. Transformation Analysis

- Used when data flows sequentially.
- Input → processing → output
- Converts DFD into structured charts.

2. Transaction Analysis

- Used when system behavior depends on transaction type.
- One input leads to different processing paths.

Example
Banking system (deposit, withdrawal, transfer).

Principles used

1. Abstraction

- It focuses on essential details.
- It hides unnecessary complexity.

2. Information Hiding

- Hide internal details of modules.
- Access only through interfaces.

3. Modularity

- Divide system into independent modules.
- Improves readability and maintainability.

⇒ Verification :-

- Verification is the process of evaluating software work products (such as requirements, design documents and code) to ensure that they conform to the specifications defined in the previous phase of development.
- In simple words, verification answers the question:
  Are we building the product right?
- Verification checks whether the software correctly implements the given specifications without actually executing the program.
- Verification is the static process of checking software artifacts to ensure that the output of a development phase meets the requirements imposed at the beginning of that phase.
- During software development, as system moves from higher level of abstraction to lower level of abstraction.
- Verification ensures that at each stage, the output is a correct refinement of the input from previous stage.
- Verification does not involve execution of code. It mainly concerned with correctness, completeness, and consistency of documents and designs.

⇒ Metrics :-

- Metrics are quantitative measures used to assess the quality, complexity, and maintainability of software systems.

Metrics help:

- Estimate effort and cost
- Measure complexity
- Predict maintenance difficulty
- Improve design quality

⇒ Network Metrics :-

- Network metrics analyze the structure chart or call graph of a software system to measure its architectural complexity.
- These metrics treat the system as a network of modules (nodes) connected by calls or relationships (edges).
- Network metrics are software design metrics that evaluate the complexity and quality of a system by analyzing the structure and relationships among modules in a structure chart.

Purpose of Network Metrics

- To evaluate how close the structure chart is to an ideal tree.
- To identify highly coupled modules.
- To detect poor architectural design.

Graph impurity

An ideal structure chart is a tree.

Graph impurity is defined as

Graph Impurity = e − n + 1
Graph Impurity = e − n + 1

Where

- n = no. of nodes
- e = no. of edges

Interpretation :

- Impurity = 0 → Perfect tree
- Higher impurity → Higher complexity and coupling

Types of Network Metrics

1. Degree correlation :-

- Degree correlation measures whether modules tend to connect to other modules with similar or dissimilar degrees.
- Positive degree correlation → High degree modules connect with high degree modules.
- Negative degree correlation → High degree modules connect with low degree modules.

Importance :

- Helps understand dependency patterns in the system.

---

2. Rich Club Metric :-

- The Rich club metric measures

the tendency of high connected modules to connect among themselves.

- High value → Strong interdependency among complex modules.
- Indicates poor modularity.

---

3. Joint degree distribution (JDD)

- JDD records how often pairs of nodes with degrees (K1, K2) are connected.
- Used to analyze structural dependency patterns.
- Helps identify clustering of modules.

---

4. K-Nearest Neighbours (KNN) :-

- KNN calculates the average degree of neighboring modules.
- Used instead of recording every connection.
- Simplifies structural analysis.
- Results can be plotted as linear or logarithmic graphs.

Information Flow Metrics :-

- Information flow metrics measure the complexity of a system based on the amount of information flowing between modules.
- Information flow includes:
  - Parameter passing
  - Variable access
  - Global data usage
- Information flow metrics quantify software complexity by measuring the flow of information into and out of software modules.

Concepts

Fan-In

- Fan-in of a module is the number of modules that call it.
- High fan-in → Reusable or heavily used module.

Fan-Out

- Fan-out of a module is the number of modules it calls.
- High fan-out → High dependency → Difficult to maintain.

Information flow complexity formula

Complexity = $(\text{fan-in} \times \text{fan-out})^2$

Complexity = $(\text{fan-in} \times \text{fan-out})^2$

This indicates that complexity grows rapidly as dependencies increase.

IFIN and IFOUT

Information fan-in

IFIN = Procedures calling + parameters read + global variables read

---

Information fan-out

IFOUT = Procedures called + parameters written + global variables written

---

Combined Metric

Information flow = IFIN × IFOUT

Information flow = IFIN × IFOUT