Unit-I

Software:

=: Software is an instruction [computer program] that when executed provide decide function & performance.

=: Data Structures that enables that program to adliquately manipulate information

=: Documents that describe the operation and usi of the programs.

=: Simply, software is known as combination of set of instruction, data structure of the documents.

Software Characteristics

Software is developed or engineered it is not manufactured in the classical sense it is a traditional sense.

Some similarities exist between software and hardware manufacture.

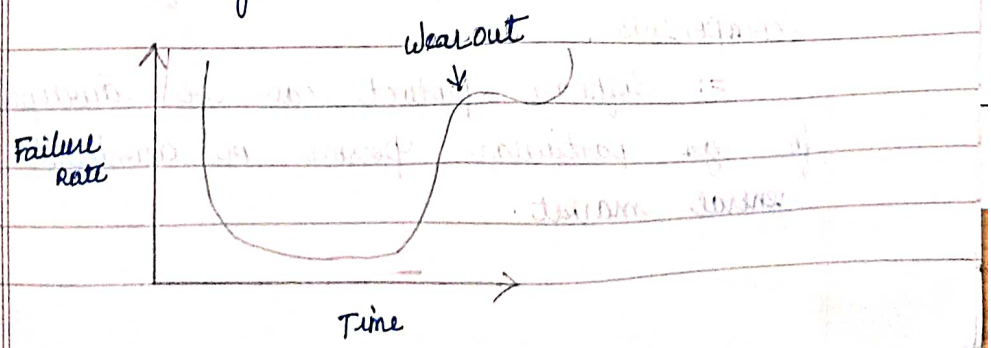
In both activities high quality can be achieved through our good design.

The manufacturing phase for hardware can introduce quality problem that are non-excistant for software.

Both activities depend on people but the work accomplished is entirely different.

ii) Software S/W does not wear out.

Here the H/W failure rate can be indicated by bath ~~tb~~ rate curve.



=: The curve indicates the hardware H/W exists high failure rate.

=: The defects are corrected & failure rate drops to steady state level for some period of time.

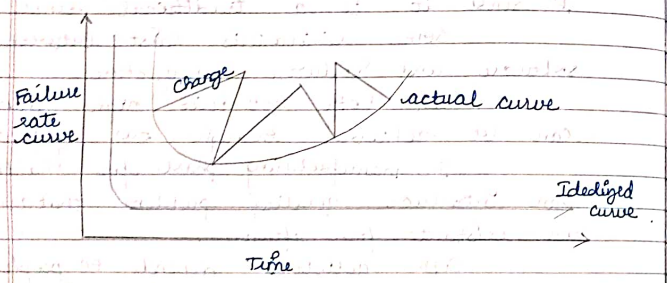
=: As time pass the failure rate raises again and so on.

=: Thus the hardware begins to "wear out".

=: Software is not affected to the environment.

=: The failure rate curve for software should take in form of Idealised curve.

=: Undiscovered defects will cause high failure rate and this are corrected indicated as flatten curve.



=: Simply, the software does not wear out. The Software can be easily modify and the software can be reproduce.

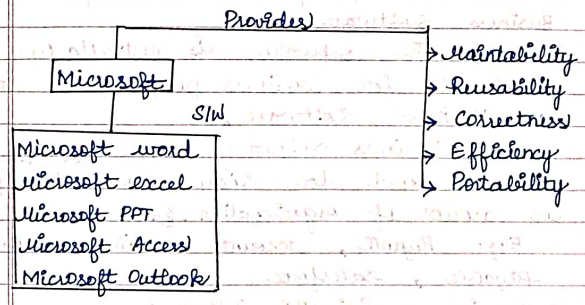
=: Most of the software is customer built rather than assemble from existing components.

=: Software product can be developed for ~~a~~ particular person or developed for general market.

=: The general characteristic of software are:

* Maintability
* Reusablity
* Correctness
* efficiency
* portability

=: Example : Micro software.



Applications of Software:

1. System S/W
2. Real time S/W
3. Business S/W
4. Engineering & Scientific S/W
5. Embedded S/W
6. PC S/W
7. Web based S/W
8. artificial Intelligence S/W.

BCA - 2nd year ①

Simathi. A ②

System Software:

=: System S/W is a collection of programs written to services other programs.

Example: OS, Compiler, Editors.

Real Time Software:

=: Software that monitors [or] Analysis or Controls the real world events as they occurs.

Example: Traffic Monitor System, Online Shopping, C-banking.

Business Software:

=: The software is particularly developed for business purpose is known as Business Software.

=: Business Software can ~~be do~~ be developed for single organization [company] or group of organization. Eg:

Eg: Payroll, accounts ~~resable or payroll~~ payable, salesforce.

Engineering & Scientific Software:

=: The software is particularly developed for engineering and scientific purposes is known as engineering and scientific software.

Eg: CAD - Computer Aided Design
Simulation Software.

Embedded Software:

=: The software is particularly developed for embedded purposes is known as embedded software.

Eg: Washing Machine
Digital Watch / Bike.

=: Embedded software which interact between software and hardware.

=: Intelligence products has been developed using embedded software.

Personal Compute Software:

=: The software only developed for personal usages over past two decaded [20 years].

Example:

Word Processing
Animation
Graphics
Spread Sheet
Browser [chrome etc...]

Web Based Software:

=: The web pages the retrieved by a browser or software that incorporates executable instruction.

Eg: HTML
Real

Artificial Intelligence Software:

=: Artificial intelligence is an area of computer science that emphasis the creation of intelligence machine that work and real like human.

=: The software is particularly developed for Artificial Intelligence.

Software Engineering:

=: Software engineering is a discipline which theories method and tools are applied to develope professional software.

=: Software engineering is a field of computer science that deals with construction of large, complex, industrial strength software system.

Various definition for Software engineering:

=: There are several definition for software they are;

Def - 1: Software engineering is a process of developing software and modified on time and with cost estimation

Def - 2: Fritz Bauer define software engineering. This software is developed in reliable work efficiently on real machines

Def - 3: Stephen Schach define software engineering, the software that is delivery on time and satisfy the user requirement.

Def 4: Boehm defines the software engineering, the application of science & mathematics by which capability of computer requirements [or] made useful to Man via programs.

Benefits of Software Engineering:

(i) Increased Reliability:

=: Components excersised in working system

(ii) Reduced Process Risk:

=: Less uncertainty in development costs.

(iii) Effective use of spicialist:

=: Reuse components instead of people.

(iv) Standards Compliance:

=: Embed standards in reusable Component

(v) Accelerated Development

=: avoid original development, hence speed-up production.

Goal of Software Engineering:

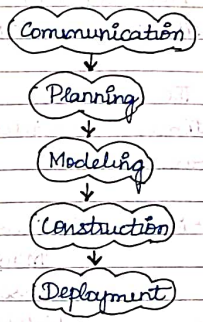
=: The various goals of software engineering are;

* Satisfy the user requirement.
* High Reliability
* Delivery on time.
* Low Maintainence Cost
* High Performance
* Reusability.

Software Process:

=: The software process is a set of related activities that leads to production of software.

=: The software process consist of activities that over carried out during the development of every software system.



Communication / Requirement Engineering:

=: Define main functionality of the software and contains around them.

=: This activity involves the gathering of software requirements from the customer and related to sub activities.

Planning:

=: This activity of planning the work require to develop the software.

=: This includes;

* Risk Management
* Listening the association Output
* Processing the scheduling for the work.

Modelling:

=: This activity is involved in modeling both the requirement and the software design.

=: So, that both the developer and customer can better understand work being carried out.

Construction:

=: This is the development of the software.

=: This activity also includes sub activity for testing software.

Deployment:

=: The software is deliver and the customer provides feedback on the software.

Example:

Communication -	Requirements -	Gathering
[Womens]	[Mens]	[Kids]
Dresses	Dresses	Toys
Accessories	Watches	Chocolate
Cosmetics	Glass	Dresses
Footwears	Footwears	Footwears
Home Applications.		

Planning:

=: Planning the work according to the requirement.

=: In planning, for this e-shopping software we should include some general categories for each requirements

General Categories:

* Cost
* Brand
* Size
* Quality
* Colours

Modelling:

=: In this modelling, we should module the design according to our planned categories.

Example:

Product :	Glass
Colour :	Blue
Size :	Normal
Design:	
Address:	
	Place Order

Construction:

=: Coding for the product should develop in this activity.

=: Test ~~in~~ the coding via the order is placed correcting or not.

=: If any error is occured, the debug the error according to the product.

Deployment:

=: After the product is ready to delivery deploy the product to client and get feedback from the client

Software Process Model:

=: For accomplishing and efficient Software project we need to ~~tde~~ track the following main five steps:

1. Communication:

=: In this stage, we have to collect or gather to requirements for our project.

2. Planning:

After collecting the appropriate requirements we have to plan the design based on requirement.

3. Modelling :

=: In this stage, we have To execute the plan and design our work.

4. Construction :

=: after testing and recoding the work we have to deliver the project to client reclaim the feedback.

=: along with ~~other~~ above 5 activities we have some extra activities called umbrella activities. That will help to clear our profit more efficient.

Umbrella activities :

* Project Tracking & Control
* Team RN
* Risk Management
* Quality Assurance
* Configuration Management.

Project Tracking & Control :

=: The monitoring and control process group helps to track the project process with purpose of insuring process is on track to meet the project objective

Risk Management :

=: Risk Management plan ~~emphas~~ explaines how to management the risk associated with the project. It also ensures that all the ~~excepted~~ expected risk are identify, monitor and control.

Quality Assurance :

=: The quality assurance plan is developed during the inception face. It helps to assure the quality of the artificial process and product.

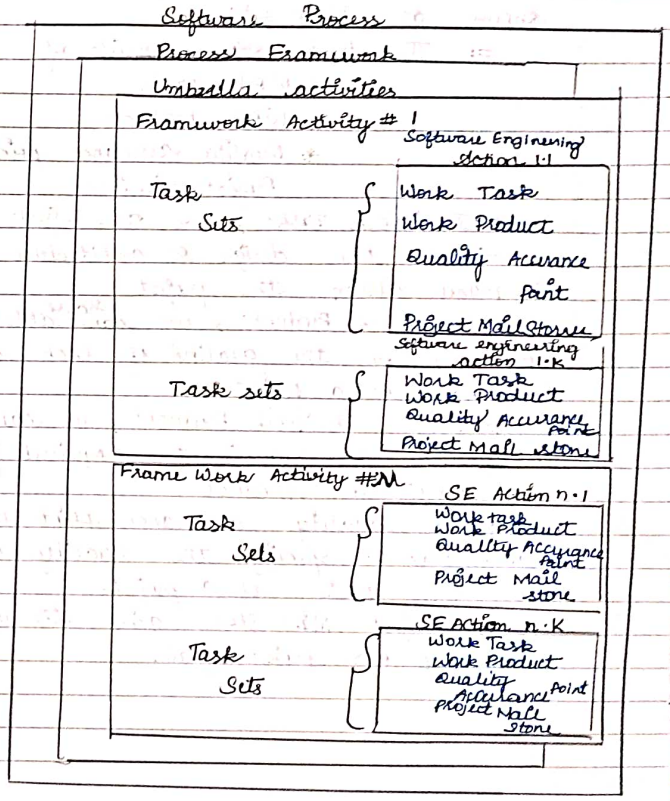
Configuration Management :

=: In this management, we are able to configrate structure of product and we can easily manage easily structure project of process.

Technical Review :

Technical reviewers in various stages here we are just presenting updates of the project in every stages.

Framework Activity / Software Process Framework



Explanation:

=: The above chart represent the activities including the umbrella activities through the framework activities according to the project.

=: An umbrella activities are comes under the process of framework.

=: In umbrella activities we are having 2 framework activities. ~~activi~~

=: In each framework activity we have 2 sets of software engineer action known as 'Task sets'.

=: The task set consist of

* Work Task
* Work product
* Quality assurance point
  Project milestone.

* In 'Work Task', we are just mentioned the design or architecture by the many design the project.

~~In 'Work Product', we can have able to spif spicify the quality of each and every parts of a project.~~

* 'Project In 'Work Product', we can able to spicify the requirement required for the particular product.

* In quality assurance point we have ~~the~~ to specify the quality of each and every parts of a project.

* 'Project Mile Stone gives the upgrade steps to our project goal.

Completed
26/12/22

UNIT-2

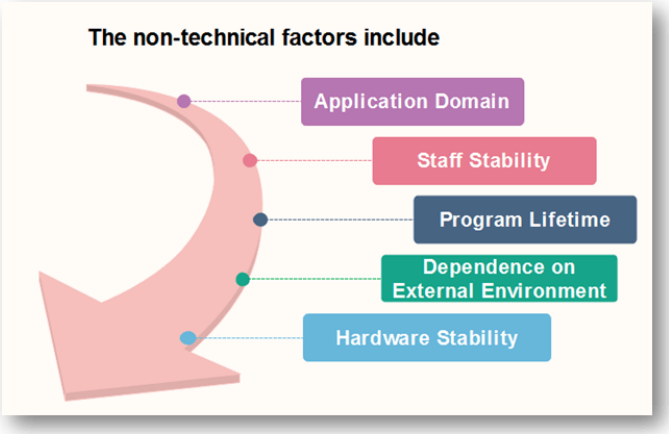
Software Maintenance Cost Factors

There are two types of cost factors involved in software maintenance.

These are

- Non-Technical Factors
- Technical Factors

Non-Technical Factors



1. Application Domain

- If the application of the program is defined and well understood, the system requirements may be definitive and maintenance due to changing needs minimized.
- If the form is entirely new, it is likely that the initial conditions will be modified frequently, as user gain experience with the system.

2. Staff Stability

- It is simple for the original writer of a program to understand and change an application rather than some other person who must understand the program by the study of the reports and code listing.
- If the implementation of a system also maintains that systems, maintenance costs will reduce.
- In practice, the feature of the programming profession is such that persons change jobs regularly. It is unusual for one user to develop and maintain an application throughout its useful life.

3. Program Lifetime

- Programs become obsolete when the program becomes obsolete, or their original hardware is replaced, and conversion costs exceed rewriting costs.

4. Dependence on External Environment

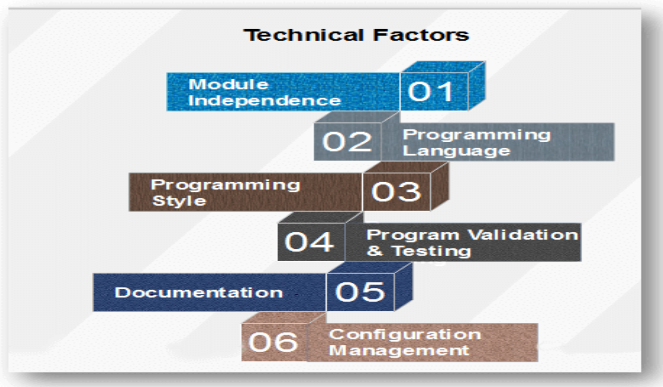
- If an application is dependent on its external environment, it must be modified as the climate changes.
- For example:
- Changes in a taxation system might need payroll, accounting, and stock control programs to be modified.
- Taxation changes are nearly frequent, and maintenance costs for these programs are associated with the frequency of these changes.
- A program used in mathematical applications does not typically depend on humans changing the assumptions on which the program is based.

5. Hardware Stability

- If an application is designed to operate on a specific hardware configuration and that configuration does not changes during the program's lifetime, no maintenance costs due to hardware changes will be incurred.
- Hardware developments are so increased that this situation is rare.
- The application must be changed to use new hardware that replaces obsolete equipment.

Technical Factors

Technical Factors include the following:



Module Independence

It should be possible to change one program unit of a system without affecting any other unit.

Programming Language

Programs written in a high-level programming language are generally easier to understand than programs written in a low-level language.

Programming Style

The method in which a program is written contributes to its understandability and hence, the ease with which it can be modified.

Program Validation and Testing

- Generally, more the time and effort are spent on design validation and program testing, the fewer bugs in the program and, consequently, maintenance costs resulting from bugs correction are lower.
- Maintenance costs due to bug's correction are governed by the type of fault to be repaired.
- Coding errors are generally relatively cheap to correct, design errors are more expensive as they may include the rewriting of one or more program units.
- Bugs in the software requirements are usually the most expensive to correct because of the drastic design which is generally involved.

Documentation

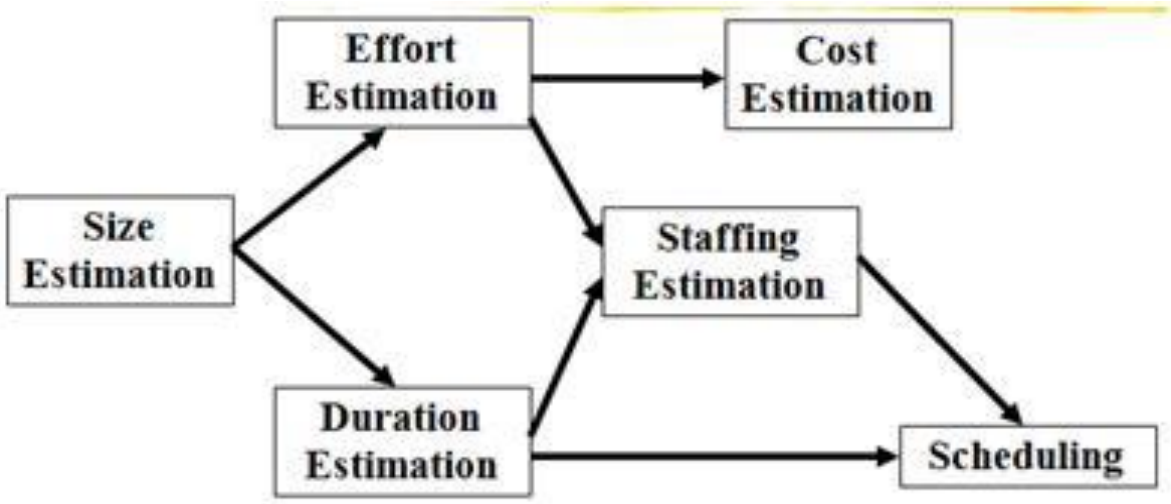
- If a program is supported by clear, complete yet concise documentation, the functions of understanding the application can be associatively straight-forward.
- Program maintenance costs tends to be less for well-reported systems than for the system supplied with inadequate or incomplete documentation.

SOFTWARE COST ESTIMATION TECHNIQUES:

Software Cost Estimation

- Determine size of the product.
- From the size estimate,
  *Determine the effort needed.
- From the effort estimate,

  *Determine project duration, and cost



Three main approaches to estimation:

- Empirical
- Heuristic
- Analytical

Software Cost Estimation Techniques

Empirical techniques:

- An educated guess based on past experience.

Heuristic techniques:

- Assume that the characteristics to be estimated can be expressed in terms of

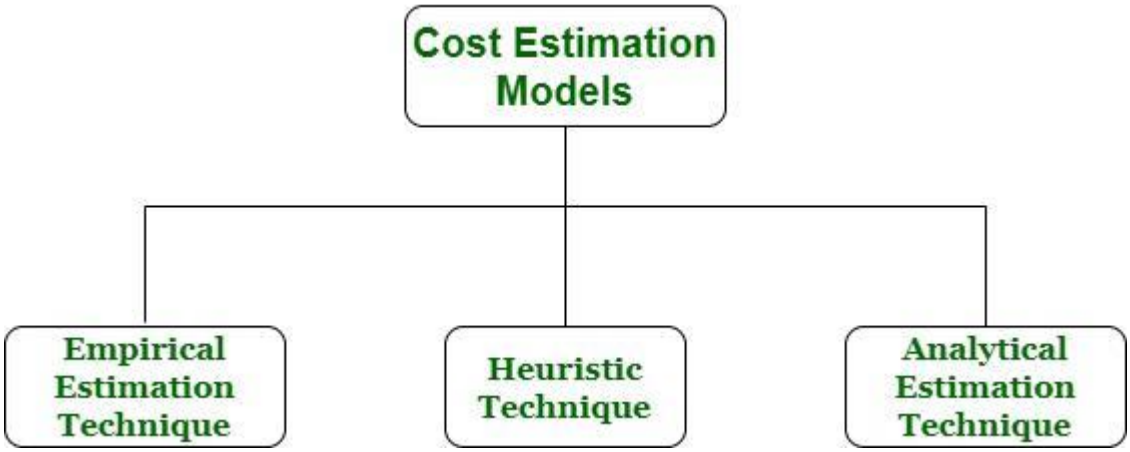
Some mathematical expression.

Analytical techniques:

- Derive the required results starting from certain simple assumptions.

Cost Estimation Models in Software Engineering:

- Cost estimation simply means a technique that is used to find out the cost estimates. The cost estimate is the financial spend that is done on the efforts to develop and test software in Software Engineering. Cost estimation models are some mathematical algorithms or parametric equations that are used to estimate the cost of a product or a project.
- Various techniques or models are available for cost estimation, also known as Cost Estimation Models as shown below :



1. *Empirical Estimation Technique :*

- Empirical estimation is a technique or model in which empirically derived formulas are used for predicting the data that are a required and essential part of the software project planning step. These techniques are usually based on the data that is collected previously from a project and also based on some guesses, prior experience with the development of similar types of projects, and assumptions. It uses the size of the software to estimate the effort.
- In this technique, an educated guess of project parameters is made. Hence, these models are based on common sense. However, as there are many activities involved in empirical estimation techniques, this technique is formalized. For example Delphi technique and Expert Judgement technique.

2. *Heuristic Technique :*

- Heuristic word is derived from a Greek word that means “to discover”. The heuristic technique is a technique or model that is used for solving problems, learning, or discovery in the practical methods which are used for achieving immediate goals. These techniques are flexible and simple for taking quick decisions through shortcuts and good enough calculations, most probably when working with complex data. But the decisions that are made using this technique are necessary to be optimal.
- In this technique, the relationship among different project parameters is expressed using mathematical equations. The popular heuristic technique is given by Constructive Cost Model (COCOMO). This technique is also used to increase or speed up the analysis and investment decisions.

3. *Analytical Estimation Technique* :

- Analytical estimation is a type of technique that is used to measure work. In this technique, firstly the task is divided or broken down into its basic component operations or elements for analyzing. Second, if the standard time is available from some other source, then these sources are applied to each element or component of work.

- Third, if there is no such time available, then the work is estimated based on the experience of the work. In this technique, results are derived by making certain basic assumptions about the project. Hence, the analytical estimation technique has some scientific basis. Halstead's software science is based on an analytical estimation model.

STAFFING LEVEL ESTIMATION:

- Once the effort required to develop a software has beendetermined, it is necessary to determine the staffingrequirement for the project
- Putnam first studied the problem of what should be aproper staffing pattern for software projects. HeextendedtheworkofNordenwhohadearlierinvestigatedthestaffingpatternofresearchanddevelopment (R&D) type of projects.
- In order to appreciate the staffing pattern of softwareprojects,Norden'sandPutnam'sresultsmustbeunderstood.

Norden's Work:

- Norden studied the staffing patterns of several R & D projects.
- He found that the staffing pattern can be approximated by theRayleighdistribution curve.
- Norden represented the Rayleigh curve by the following equation:
- $E = K/t^2d * t * e-t^2 / 2 t^2 d$
- Where,
- E is the effort required at time t. E is an indication of the number ofengineers (or the staffing level) at any particular time during the durationof the project
- K is the area under the curve, and
- tdis the time at which the curve attains its maximum value.
- It must be remembered that the results of Norden are applicable togeneral R & D projects and were not meant to model the staffing pattern

ESTIMATION SOFTWARE MAINTANACE COSTS:

How to Estimate Software Maintenance Cost

Estimating software maintenance cost is often very complicated because of the numerous factors and components one needs to keep in mind. The first thing you have to do is make a clear plan of the actions that must be performed during the maintenance process. Then the IT department does the analytics. They will estimate the time necessary to complete each task and give you developer rates. The final step is making calculations based on the information received. If done properly, you'll get your software maintenance cost.

Some time ago we released a post about Android app development costs. Feel free to read it!
There are some things you should consider when making a plan for costs estimation:

- All the fees involved in hosting services, purchasing and prolonging software licenses, and any SaaS accounts;
- iOS and Android account fees, if you are developing a mobile application;
- The cost of debugging: the number of bugs times the cost of one bug;
- Possible design updates;
- Plugin updates, third-party software support;
- Code refactoring.

It's not the full list of the things to keep in mind, but it does contain the most crucial points you can't miss under any circumstances.

Now imagine that we have a mobile app to maintain. Let's try to calculate the approximate budget.

- An app hosting server can cost anywhere from $70 per month up to $320 per month, which largely depends on such factors as the app's content, the number of active users, and its projected growth;
- The fees for iOS and Android accounts are 99$ and 25$ respectively;
- Software licenses can be free (say, for using iTunes) or cost up to tens of thousands of dollars (say, for using EDA tools);
- SaaS account fees can be up to a thousand dollars;
- Let's say your developers' rate is 55$/hour and the time needed to eliminate one bug is 1 hour. If the average amount of bugs per month is 20, you will need to spend 1100$ on debugging;
- According to Upwork, the average hourly rate of a designer is 25$. For example, a quick design update requires about half a week of work. So, designer services will cost you 500$ a month;
- In a perfect world, there is no need for code refactoring because everything is ready long before the app's release. But in reality, there are situations when developers have to write code very quickly. As a result, the code often ends up being pretty messy, so it requires some refactoring. A developer needs up to 3 hours to finish this task. According to the mentioned rates, it will cost about 150$.

Of course, other expenses may arise. In fact, there are so many possible unforeseen costs, that it's really hard to consider all of them. There are several main factors that make up the total software maintenance cost. Let's make some calculations with those factors.

bamboo agile

FACTOR	PRICE, $
Hosting	200
iOS + Android fees	99+25
Software licenses and SaaS accounts	3500
Debugging	1100
Design updates	500
Code refactoring	150
TOTAL	**5574** + **unaccounted expenses**

THE SOFTWARE REQUIREMENTS SPECIFICATION:

1. **SOFTWARE REQUIREMENTS SPECIFICATION (SESSION 1 – SESSION 12)**

   Already we discussed in class room.

2. **FORMAL SPECIFICATION TECHNIQUES**

   2.1 RELATIONAL NOTATION

   2.2 STATE-ORIENTED NOTATIONS

3. **LANGUAGES AND PROCESSORS FOR REQUIREMENT SPECIFICATION**

   3.1 PSL/PSA(Problem Statement Language/Analyzer)

   3.2 RSL/REVS(Requirement Statement Language/Requirement Engineer Validation System)

   3.3 SADT(Structured Analysis and Design Technique)

   3.4 SSA(structured System Analysis)

   3.5 Gist

FORMAL SPEFICATION TECHNIQUES:

2.Formal Specifications Techniques

This techniques used for **mathematical** equation. In this techniques we have two notations.

2.1. Relational Notations

It is based on the concerts of **entities and attributes.**

Entities are **named elements** and **attributes** are **relations of entities**.

In this we have

2.1.1 Implicit Equation

2.1.2 Recurrence Equation

2.1.3 Algebraic Axioms

2.1.4 Regular Expressions

2.1.1 Implicit Equation

Implicit equations specify the properties of a solution **without stating a solution method**. Matrix inversion is specified as follows

$$M \times M' = I + E$$

Matrix inversion has the property that the original matrix (M) multiplied by its inverse (M1) yield an identify matrix, I denotes the identify matrix and E specifies allowable computational errors

2.1.2 Recurrence Relations

Its consists of an **initial part** called basis and one or more recursive parts

Example: Fibonacci Series

$F(0) = 0$

$F(l)=l$

$F(N) = F(N-1) + F(N-2)$ for $N>1$

2.1.3 Algebraic Axioms

It is used to specify the **properties** of abstract data types

Example: Stack

(Stk is of type STACK, itm is of type ITEM)

1. EMPTY(NEW) = true
2. EMPTY(PUSH(Stk,itm)) = false .
3. POP(NEW) = error
4. TOP(NEW) = error
5. POP(PUSH(stk,itm)) = stk
6. TOP(PUSH(stk,itm)) = item

2.1.4 Regular Expressions

The rules for forming regular expressions are as follows.

Axioms: The basis symbols in the alphabet of interest form regular expressions.

Alternation: If Rl and R2 are regular expressions, then (R1/R2) is a regular expression.

Composition: If Rl and R2 are regular expressions, then (R1.R2) is a RE

Closure: If Rl is a regular expressions, then (Rl)* is a regular expression.

Completeness: Nothing else is a regular expression.

Example: (a(b/c)) denotes {ab,ac}

2.2 STATE ORIENTED NOTATION

2.2.1 Decision Table

2.2.2 Events Tables

2.2.3 Transition Tables

2.2.4 Finite State Mechanisms

2.2.5 Petri Nets

2.2.1 Decision tables

Decision tables are widely used in **data processing application**. A decision table is segmented into **four quadrants**, condition state, condition entry, action states and action entry

Example:

Conditions	**R1**	**R2**	**R3**
Withdrawal Amount <= Balance	T	F	F
Credit granted	-	T	F
Actions			
Withdrawal granted	T	T	F

2.2.2 Event tables

specify actions to be taken when **events occur under different set of conditions**. Event tables are viewed as two- dimensional tables or of higher dimension.

Example:

Staff	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Designer	4	4	3	3	2	2	1
Developer	0	0	1	2	4	4	3
Tester	0	0	0	0	2	2	2
Total	4	4	4	5	8	8	6

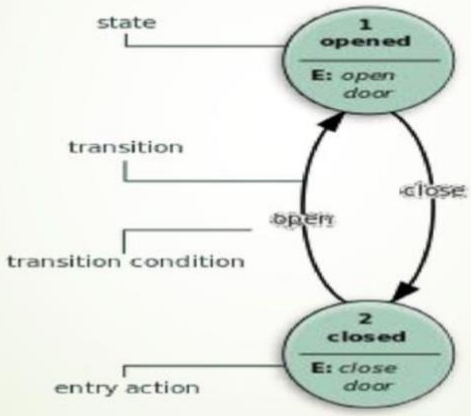
2.2.3 Transition Table

Transition Tables are used to specify **changes in the state** of a system as a function of next state

Current State	Current Input	
INPUT	A	B
S0	S0	S1
S1	S1 (next state)	S0

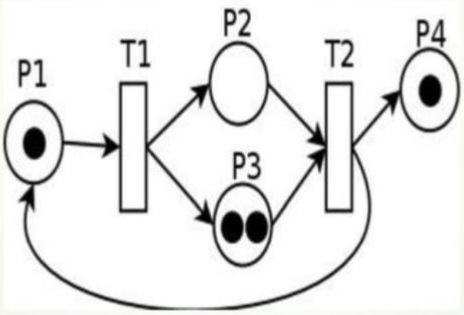
2 .2.4 Finite State Mechanisms:

utilize **data flow diagrams** in which the data streams are specified using **regular expressions** and the actions in the **processing nodes** are specified using transition labels.



2.2.5 Petri Net

It represent the **technique and systematic** methods have been developed for synthesizing and **analysing** petri nets. Petri nets were invented to **overcome the limitations of finite state mechanisms in specifying parallelism.**



LANGUAGES AND PROCESSORS FOR REQUIREMENTS SPECIFICATION:

3. LANGUAGES AND PROCESSORS FOR REQUIREMENT SPECIFICATION

3.1 PSL/PSA(Problem Statement Language/Analyzer)

Originally developed for **data processing applications**. Widely used in other applications.

In PSL, system descriptions can be divided into eight major aspects:

System input/output flow.
System structure
Data structure
Data derivation
System size and volume
System dynamics
System properties
Project management

3.2 RSL/REVS(Requirement Statement Language/Requirement Engineer Validation System)

Real time process control systems

Major Components are

1.A translator for RSL.

2.A centralized database, the abstract semantic model(ASSM)

3.A set of automated tools for processing information in ASSM.

3.3 SADT(Structured Analysis and Design Technique)

Interconnection structure of any **large, complex system.**

Not restricted to software systems.

3.4 SSA(Structured System Analysis)

Two similar versions of SSA are

Gane and Sarson version used in **data processing applications that have data base requirements**.

DeMarco version suited to **data flow analysis of software systems.**

3.5 Gist

Object-oriented **specification and design.**

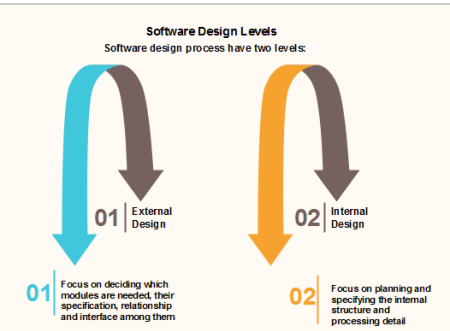
Refinement of **specifications into source code**.

UNIT-3

SOFTWARE DESIGN:

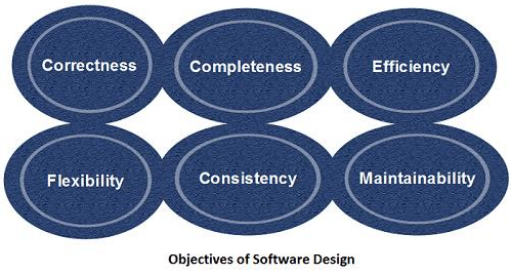
Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

The software design phase is the first step in **SDLC (Software Design Life Cycle)**, which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.



Objectives of Software Design

Following are the purposes of Software design:

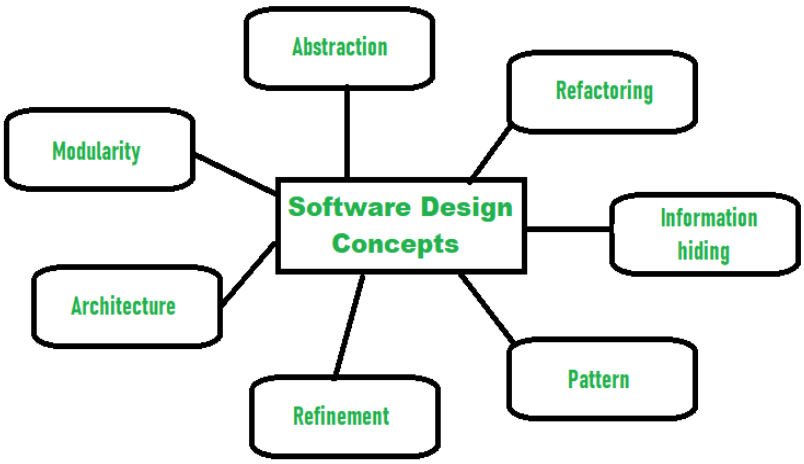


1. **Correctness:** Software design should be correct as per requirement.
2. **Completeness:** The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency:** Resources should be used efficiently by the program.
4. **Flexibility:** Able to modify on changing needs.
5. **Consistency:** There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

FUNDAMENTAL DESIGN CONCEPT:

Software Design Concepts:

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:



The following **points should be considered while designing Software:**

1. **Abstraction- hide Irrelevant data**
   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity- subdivide the system**
   Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.

3. **Architecture- design a structure of something**
   Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement- removes impurities**
   Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern- a repeated form**
   The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding- hide the information**
   Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.
7. **Refactoring- reconstruct something**
   Refactoring simply means reconstructing something in such a way that it does not affect the behaviour of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behaviour or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behaviour of the design and improves the internal structure".

Different levels of Software Design:

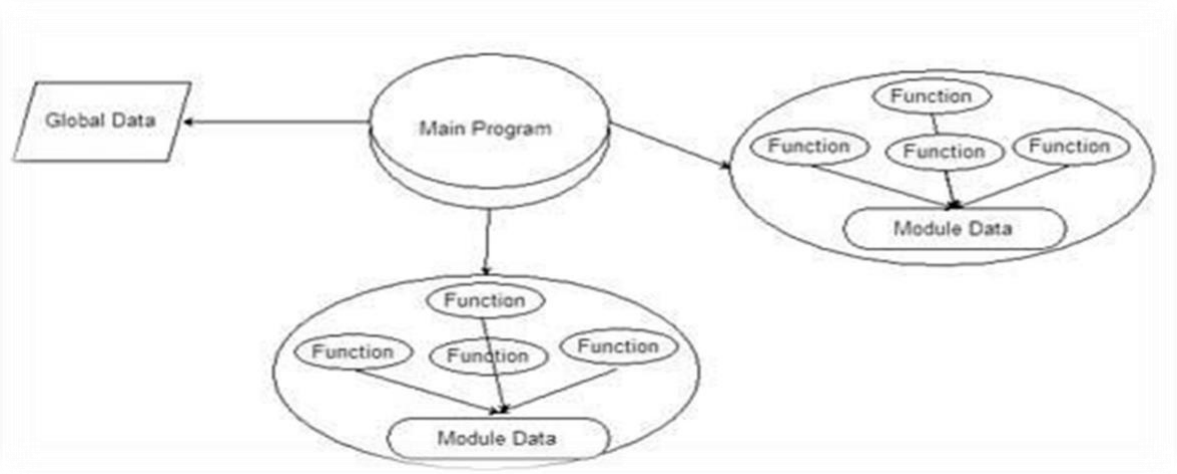
There are three different levels of software design. They are:

1. **Architectural Design:**
   The architecture of a system can be viewed as the overall structure of the system & the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.

2. **Preliminary or high-level design:**
   Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.

3. **Detailed design:**
   Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

MODULES AND MODULARIZATION CRITERIA:



- ***Modular design, or "modularity in design", is a design approach that subdivides a system into smaller parts called modules or skids, that can be independently created and then used in different systems. A modular system can be characterized by functional partitioning into discrete scalable, reusable modules; rigorous use of well-defined modular interfaces; and making use of industry standards for interfaces.***
- Besides reduction in cost (due to less customization, and shorter learning time), and flexibility in design, modularity offers other benefits such as augmentation (adding new solution by merely plugging in a new module), and exclusion.
- **Examples of modular systems** are cars, computers, process systems, solar panels and wind turbines, elevators and modular buildings.
- **Earlier examples include looms, railroad signalling systems, telephone exchanges, pipe organs and electric** power distribution systems.
- Computers use modularity to overcome changing customer demands and to make the manufacturing process more adaptive to change (see modular programming).
- Modular design is an attempt to combine the advantages of standardization (high volume normally equals low manufacturing costs) with those of customization.
- A downside to modularity (and this depends on the extent of modularity) is that low quality modular systems are not optimized for performance. This is usually due to the cost of putting up interfaces between modules.
- Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
- Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

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

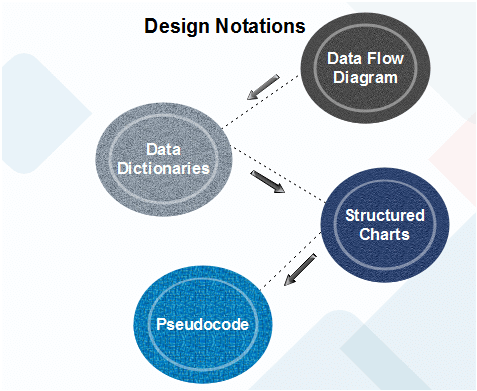
DESIGN NOTATION:

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-

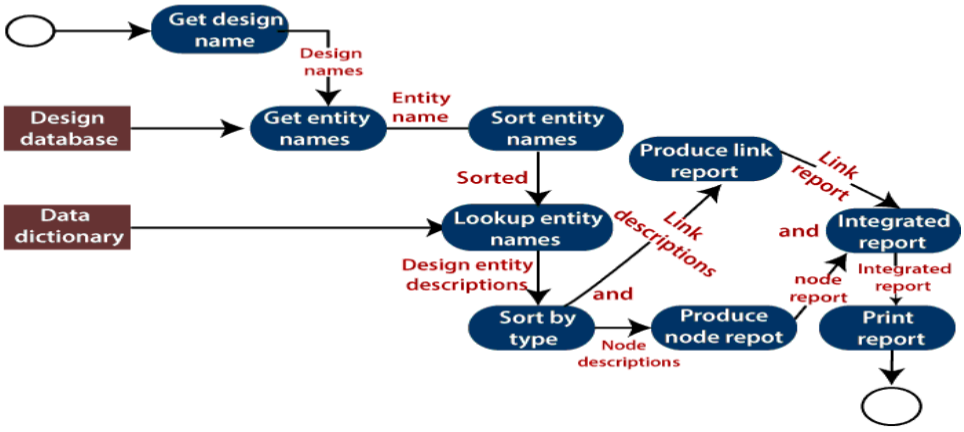
end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

- 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*.

gross Pay = regular Pay + overtime Pay

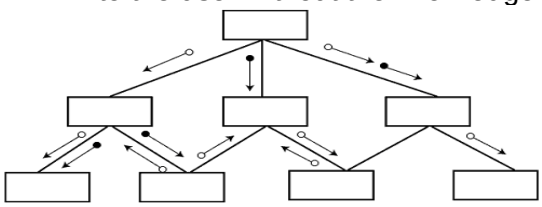
- For the smallest units of data elements, the data dictionary lists their name and their type.

- 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.

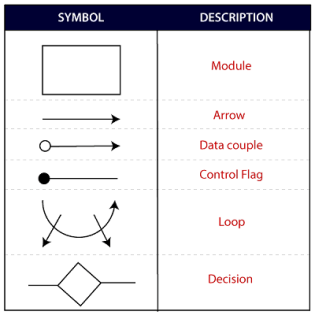


Hierarchical format of a structure chart

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.

DESIGN TECHNIQUES:

The design process involves developing a conceptual view of the system, establishing system structure, identifying data streams and data stores, decomposing high level functions into sub functions, establishing relationships and interconnections among components, developing concrete data representations, and specifying algorithmic details. Software design is a creative activity.

As with all creative processes, the system is decomposed into subsystems and more consideration is given to specific issues. Backtracking is fundamental to top-down design. In the bottom – up approach to software design, the designer first attempts to identify a set of primitive objects, actions, and relationships that will provide a basis for problem solution. Higher-level concepts are then formulated in terms of the primitives. The bottom-up strategy requires the designer to combine features provided by the implementation language into more sophisticated entities.

Stepwise Refinement: Stepwise refinement is a top-down technique for decomposing a system from high-level specifications into more elementary levels. Stepwise refinement involves the following activities:
Decomposing design decisions to elementary levels.

1. Isolating design aspects that are not truly interdependent.
2. Postponing decisions concerning representation details as long as possible.
3. Carefully demonstrating that each successive step in the refinement process is a faithful expansion of previous steps. The major benefits of stepwise refinement as a design technique are:

Top-down decomposition.
Incremental addition of detail
Postponement of design decisions

4. Continual verification of consistency.

Levels of Abstraction: Levels of abstraction was originally described by Dijkstra as a bottom-up design technique. In Dijkstra's system each level of abstraction is composed of a group of related functions, some of which are externally visible and some of which are internal to the level. Internal functions are hidden from other levels; they can only be invoked by functions on the same level. The internal functions are used to perform tasks common to the work being performed on that level of abstraction. Each level of abstraction performs a set of services for the functions on the next higher level of abstraction.

Structured Design: Structured design was developed by Constantine as a top-down technique for architectural design of software system. The basic approach in structured design is systematic conversion of data flow diagrams into structure charts. Design heuristics such as coupling and cohesion are used to guide the design process.

The first step in structured design is review and refinement of the data flow diagram developed during requirements definition and external design. The second step is to determine whether the system is transform cantered or transaction-driven, and to derive a high level structure chart based on this determination.

The third step in structured design is decomposition of each subsystem using guidelines such as coupling, cohesion, information hiding, and levels of abstraction, data abstraction, and the other decomposition criteria. The primary strength of structure design is provision of a systematic method for converting data flow diagrams into top-level structure charts.

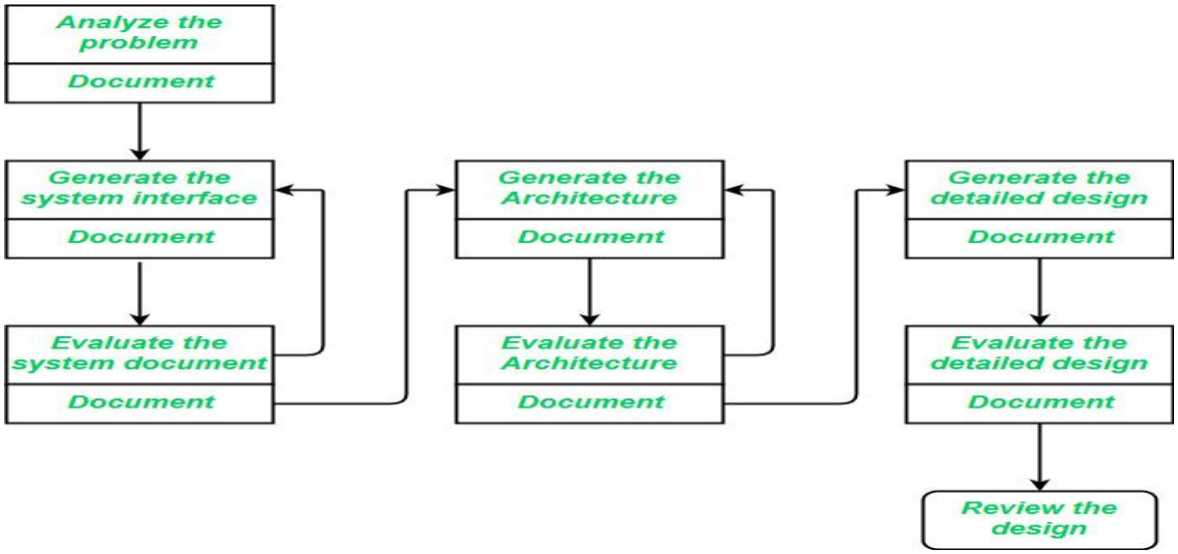
Integrated Top-Down Development: Integrated top-down development integrates design, implementation, and testing. Using integrated top-down development, design precedes top-down from the highest-level routines; they have the primary function of coordinating and sequencing the lower-level routines.

Lower-level routines may be implementation of elementary functions or they may in turn invoke more primitive routines. There is thus a hierarchical structure to a top-down system in which routines can invoke lower-level routines but cannot invoke routines on a higher level.

DETAILED DESIGN CONSIDERATIONS:

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.
The software design process can be divided into the following three levels of phases of design:

1. Interface Design
2. Architectural Design
3. ***Detailed Design***



Interface Design:
Interface design is the specification of the interaction between a system and its environment. this phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called *agents*.

Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification on the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

Architectural Design:

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.

<u>An issue in architectural design includes:</u>

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between components.

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

Detailed Design:

Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

REAL-TIME AND DISTRIBUTED SYSTEM DESIGN:

REAL TIME AND DISTRIBUTED DESIGN

- In Software Engineering, many of the popular design "methodologies" were developed as design techniques for application programs, operating systems and utility programs(compilers, editors, loaders, etc.).
- These methods support concepts such as hierarchical decomposition, modularity, information hiding, and data abstraction.
- These design concepts are important for real time and distributed systems.

- A Distributed system consists of a collection of nearly autonomous processors that communicate to achieve a coherent computing system.

- Each processor possesses a private memory, and processors communicate through an interconnection network.

ISSUES

Major issues to be addressed in designing a distributed system includes,

- ✓ Specifying the topology of the communication network
- ✓ Establishing rules for accessing the shared communication channel
- ✓ Allocating application processing functions to processing nodes in the network
- ✓ Establishing rules for process communication and synchronization.

REAL TIME SYSTEMS

- Real time systems must provide specified amounts of computation within fixed time intervals.
- Real time systems typically sense and control external devices, respond to external events, and share processing time between multiple tasks.
- Real time systems often form distributed networks; local processors may be associated with sensing devices and actuators.

- A Real time network for process control may consist of several minicomputers and microcomputers connected to one or more large processors.
- Each small processor may be connected to a cluster of real time devices.
- Decomposition criteria for distributed real time systems include the need to maintain process simplicity and to minimize inter-process communication bandwidths by communicating simple processed messages rather than raw data.

NOTATIONS

Several notations have been developed to support analysis and design of real time and distributed systems.

They include RSL, SDLP, NPN, communication ports

State Oriented

Most of these notations are state-oriented because a real-time system can be thought of as possessing a(perhaps large) number of processing states, with the transitions between states triggered by external stimuli.

Petri Nets

- Petri nets are a fundamental state-oriented notation that can be used to specify requirements and high level design of real-time and distributed systems.
- Petri nets were developed because traditional finite state mechanisms are not adequate for specifying parallel and concurrent system properties.

SUMMARY OF REAL TIME SYSTEMS

- The traditional considerations of hierarchy, information hiding, and modularity are important concepts for the design of real-time systems.
- However, these concepts are typically applied to the individual components of a real-time system.
- Higher-level issues of networking, performance, and reliability must be analyzed and designed before the component nodes or processes are developed.

TEST PLANS:

TEST PLANS

- The test plan is an important, but often overlooked , product of software designs. A test plan prescribes various kinds of activities that will be performed to demonstrate that the software product meets its requirements.
- The test plan specifies the objectives of testing. The test completion criteria the system integration plan methods to be used on particular modules and particular test cases to be used.

FOUR TYPES OF TESTS THAT A SOFTWARE PRODUCT MUST SATISFY.

- Functional Test
- Performance Test
- Stress Test
- Structural Test

FUNCTIONAL TEST

- Functional tests and performance tests are based on the requirements specifications; they are designed to demonstrate that the system satisfies its requirements.
- Functional test cases specify typical operating conditions, typing input values, and typical expected results.
- Functional tests should also be designed to test boundary conditions just inside and just beyond the boundaries.

e.g.., square root of negative numbers, inversion of one-by –one matrices ,etc.

PERFORMANCE TEST

- Performance tests should be designed to verify response time, execution time, throughput, primary and secondary memory utilization and traffic rates on data channels and communication links.
- Each functional and performance test should specify the machine configuration, assumptions concerning the system status for the test case, the requirements being tested , the test inputs and the expected results.

STRESS TEST

- Stress tests are designed to overload a system in various ways, such as attempting to sign on more than the maximum allowed number of terminals, processing more than the allowed number of identifiers or static levels, or disconnecting a communication link.
- The purpose of stress testing are to determine the limitations of the system and, when the system fails, to determine the manner in which the failure is manifest.
- Stress tests can provide valuable insight concerning the strengths and weaknesses of a system.
- Stress tests are derived from the requirements, the design, and the hunches and intuitions of the designers.

STRUCTURAL TEST

- Structural tests are concerned with examining the internal processing logic of a software system.
- The particular routines called and the logical paths traversed through the routines are the object of interest.
- The goal of structural testing is to traverse a specified number of paths through each routines in the system to establish thoroughness of testing.

MILESTONES, WALKTHROUGH, INSPECTIONS:

MILESTONES

- One of the most important aspects of a systematic approach to software development is the resulting visibility of the evolving product.
- The system becomes explicit, tangible and accessible.
- The evolving User's Manual, architectural design specification, detailed design specification and the test plan.

MAJOR PARTS OF MILESTONE

Two major milestone during software design

(i) Preliminary Design Review(PDR)

(ii) Critical Design Review(CDR)

The PDR is typically held near the end of architectural design and prior to detailed design.

CDR occurs at the end of detailed design and prior to implementation.

Functional characteristics, performance attributes, external interfaces, user dialogues, report formats, exception conditions and exception handling product subsets, future enhancement to the product should all be reviewed during the PDR.

WALKTHROUGHS

- A structure walkthrough is an in-depth, technical review of some aspect of a software system.
- Walkthrough can be used at any time, during any phase of a software project.
- Thus, all or any part of the software requirement ,the architecture design specification, the detailed design specification.
- The test plan, the code ,supporting document, or a proposed maintenance modification can be reviewed at any stage of evolving
- Walkthrough term consists of four to six people .

INSPECTION

- Design inspections are conducted by terms of trained inspectors who work from checklist of terms to examine
- A typical design inspection term consists of a moderator/secretary, a designer, an implementer, and a tester.
- The designer ,implementer ,and tester may or may not be the people responsible for actual design, implement and testing of product inspection.
- Term member are trained for their specific roles and typically conduct two 2-hour per day.
- In one experiment 67 percent of the error.
- Another experimenter reported 70 percent error.

IMPLEMENTATION ISSUSES:

- During software engineering process, the critical stage is system implementation. At this stage the executable version of the software is created.

- Some of the important implementation issues are:

1. Reuse
2. Configuration Management
3. Host-target development

Reuse

- Making use of existing code during the software development is called reuse .
- The only significant reuse is reuse of functions and objects in programming libraries.
- Software reuse is possible at different levels. These reuse levels are as follows:

1. **Abstraction level:** At this level of reuse the software is not directly reused. Instead, the knowledge of successful abstractions is used in the design of the intended software system.
2. **Object level:** At this level of reuse , instead of writing the code, the ready objects from the library are directly used.
3. **Component level:** At this level the components can be reused. The components are basically the collection of objects and object classes that can be reused on the application software.
4. **System level:** At this level entire application system can be reused.

Reuse costs

1. The time spent in looking for software to reuse and analyzing it to check whether the identified software meets the requirements or not.
2. The required reusable software can be bought. For large off-the-shelf systems this cost can be high.
3. The cost can be added up while adapting and reconfiguring the reusable software components in order to fulfill the requirements of the system being developed.
4. The cost also gets increased when integrating reusable software elements with each other and with the new code being developed for your system.

Configuration management

1. The process of managing the changes in the software system is called configuration management.

2. The aim of configuration management is to support system integration process so that all developers can access the project code and documents in controlled way.

Configuration management activities

1. Version management: by version management, it is possible to keep track of all the versions and corresponding changes made in those versions. Due to version management there can be co-ordination in the programs developed by various programmers.

2. System integration: System integration helps the developers to define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
3. Problem tracking: During this configuration management activity support is provided to report bugs and other problems, and to allow all the developers to see who is working on these problems.

Host Target Development

1. Host is the machine on which the software is developed
2. Target is the machine on which the software is run.
3. Most of the software development is host target development. The host target development is a kind of software development in which the software is developed on one computer and is run on another.

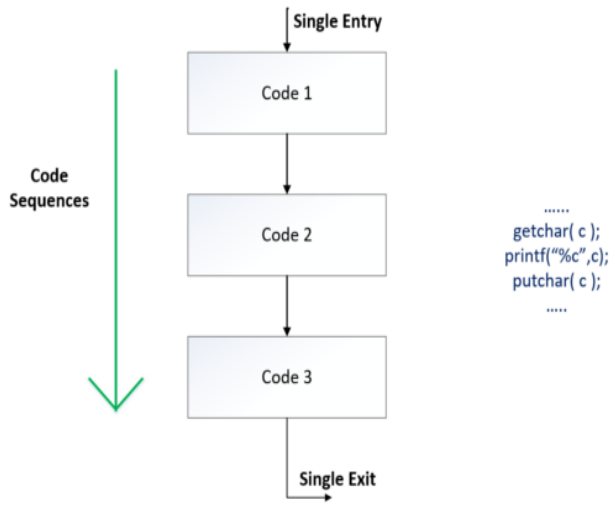
STRUCTURED CODING TECHNOIQUES:

Structured coding techniques *aids in linearizing the control flow through a computer program so that the execution sequence follows the sequence in which the code is written.* This dynamic structure of a program as it executes them resembles the static structure of the written text which improves code-readability, debugging, testing, documentation and modification of programs. The following are some of the golden rules to be considered for structured coding.

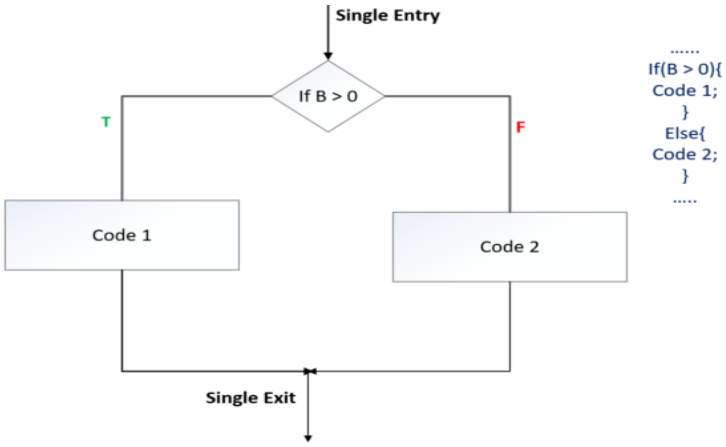
1. **Single entry, Single exit constructs**

Single entry, single exit constructs are control structures that enables us to create sequencing, selection and iteration constructs. The term single entry, single exit is used since such constructs have a single entry or single path of inflow of data and a single exit or single path of outflow. The following are the common single entry, single exit constructs –

Sequencing Construct

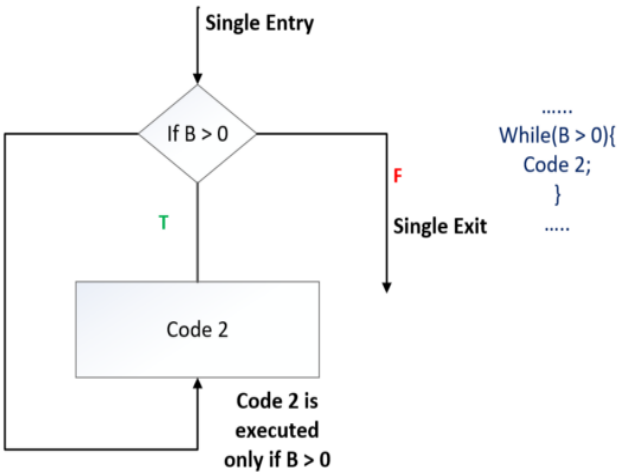


Selection Construct

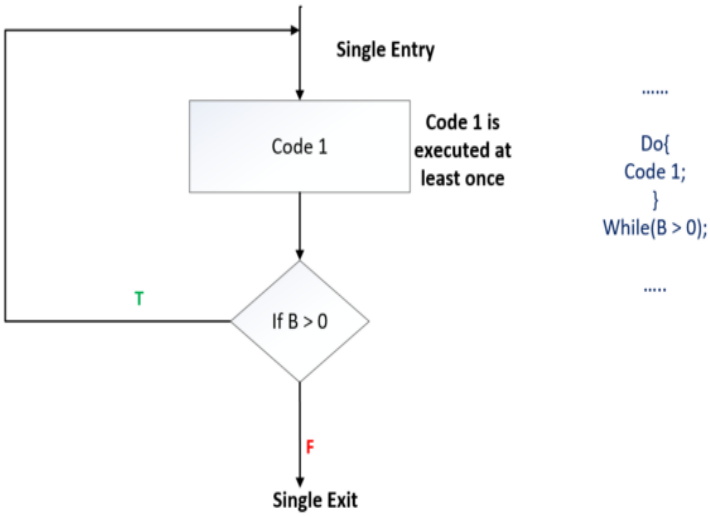


Iteration Construct

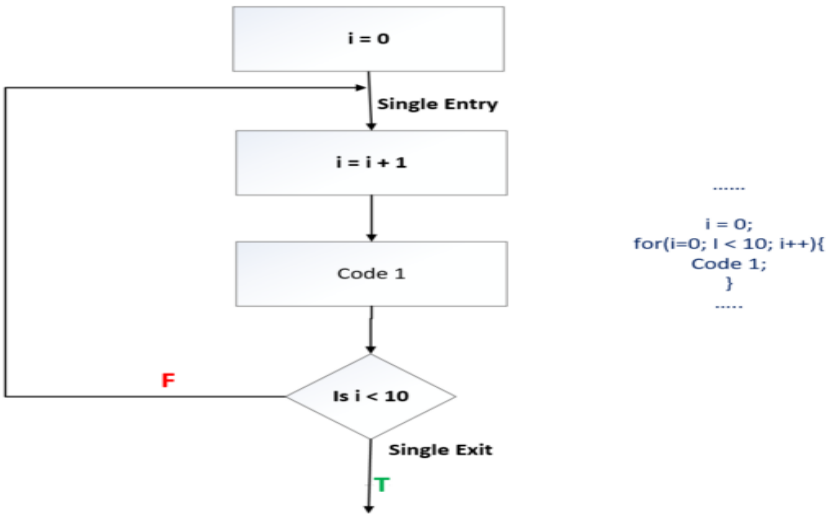
Type 1: while-do looping



Type 2: do-while looping



Type 3: for loop



In cases when we require multiple loop exits, we may apply 'go to' as some languages allow it or should redesign the algorithm to avoid multiple loop exits, which is the desired practice. Alternatively in such cases, we may use the exit or 'break' statement as follows –

```
while (cond 1){

    Code 1;

    Code 2;

    if (Cond 2){

            break;

    }
```

```
}
```

2. **Data encapsulation**

Data encapsulation is the process of encapsulating the data operations of an object from outside interference or misuse.

In object oriented languages such as C++, C# or Java, we implement data encapsulation by class abstractions.

In such a case the class itself doesn't have physical existence and is only a logical abstraction.

When a class is instantiated, it creates objects with the encapsulated data and operations.

Now, other outside groups or objects could access the internal data structure, but only the specific object.

In this way, we encapsulate data and code within each object. Again, another advantage of encapsulation is that, the data attributes or data structures are made private within a class and only the object's public functions may access and manipulate them.

Encapsulation supports abstract classes (those which cannot be instantiated but whose abstract functions may be redefined or implemented) and polymorphism (single implementation, multiple signatures).

In C++, we may encapsulate data and code within a class declaration as follows –

```
class Stack {

    private:

        int stk[MAX];

        int stktop;

    public:

        void init();

        void push(int i);
```

```
            void pop();

} stack 1;
```

Class Stack is an abstraction of data and operations as show above and stack 1 is the object that instantiates the class and which has access to the encapsulated data and code.

3. **Recursion**

A recursive programming style may be applied when the iterative processing may be indirectly performed by recursively calling the same program. Recursive algorithms may be applied, for eg:., in traversing a binary tree as follows –

```
/*Recursive C routine for inorder traversal*/



void intrav(NODEPTR){

    if(tree != NULL){

        intrav(tree->left);/*traverse left subtree*/

        printf("%d",tree->info);/*visit root*/

        intrav(tree->right);/*traverse rightsubtree*/

    }

}
```

Recursion is supported by a system provided stack to hold values of local variables, parameters and return points. Recursion cannot be performed on a routine under static memory management as in case of old Fortran or similar languages since it leads to overwriting of the original return address for the routine/method.

Recursion is not appropriate for implementing algorithms that are inherently iterative and could be implemented clearly by iteration. For eg: factorial calculation and Fibonacci operation. The reason is that there is no need for recursion as its really needed in case of binary tree traversal or backtracking algorithms used in AI.

4. **Documentation**

Structured programming constructs, good coding style, meaningful identifier names to denote user-defined data types, variable, enumerators, literals, parameters can vastly improve code readability and reduce the need for internal documentation. Some of the general guidelines for internal documentation is as follows –

- Minimize the need for embedded comments by using
  - Standard prologues
  - Structured programming constructs
  - Good Coding style
  - Descriptive names from the problem domain for user-defined data types, variables, formal parameters, enumeration literals, methods, files etc.
  - Self documenting features of the implementation language such as user-defined exceptions, user defined data types, data encapsulation etc.

- Attach comments to blocks of code that
  - perform major data manipulations
  - simulate structured control constructs
  - perform exception handling

- Use problem domain terminology in the comments
- Use blank lines, borders, and indentation to highlight comments
- Place comments to the far right to document changes and revisions
- Don't use long, involved comments to document obscure, complex code, instead rewrite the

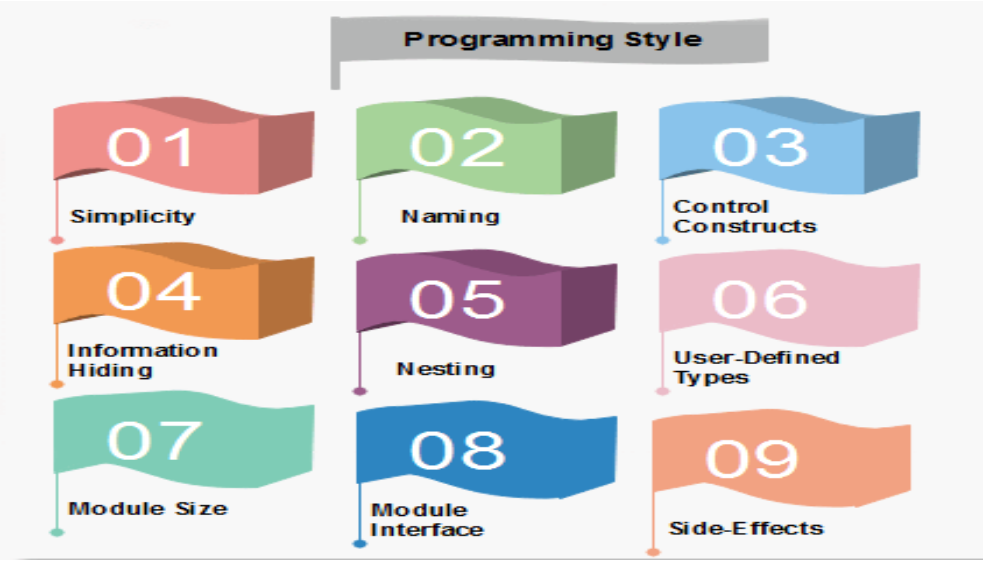
CODING STYLE:

Programming style refers to the technique used in writing the source code for a computer program. Most programming styles are designed to help programmers quickly read and understands the program as well as avoid making errors. (Older programming styles also focused on conserving screen space.) A good coding style can overcome the many deficiencies of a first programming language, while poor style can defeat the intent of an excellent language.

The goal of good programming style is to provide understandable, straightforward, elegant code. The programming style used in a various program may be derived

from the coding standards or code conventions of a company or other computing organization, as well as the preferences of the actual programmer.

Some general rules or guidelines in respect of programming style:



1. Clarity and simplicity of Expression: The programs should be designed in such a manner so that the objectives of the program is clear.

2. Naming: In a program, you are required to name the module, processes, and variable, and so on. Care should be taken that the naming style should not be cryptic and non-representative.

For Example: a = 3.14 * r * r
area of circle = 3.14 * radius * radius;

3. Control Constructs: It is desirable that as much as a possible single entry and single exit constructs used.

4. Information hiding: The information secure in the data structures should be hidden from the rest of the system where possible. Information hiding can decrease the coupling between modules and make the system more maintainable.

5. Nesting: Deep nesting of loops and conditions greatly harm the static and dynamic behavior of a program. It also becomes difficult to understand the program logic, so it is desirable to avoid deep nesting.

6. User-defined types: Make heavy use of user-defined data types like enum, class, structure, and union. These data types make your program code easy to write and easy to understand.

7. Module size: The module size should be uniform. The size of the module should not be too big or too small. If the module size is too large, it is not generally functionally cohesive. If the module size is too small, it leads to unnecessary overheads.

8. Module Interface: A module with a complex interface should be carefully examined.

9. Side-effects: When a module is invoked, it sometimes has a side effect of modifying the program state. Such side-effect should be avoided where as possible.

STANDARD AND GUIDELINES:

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.

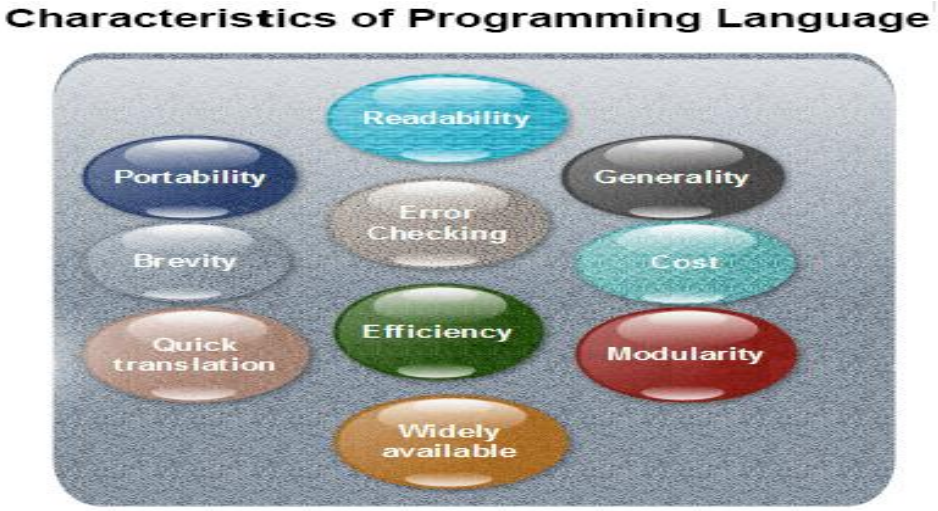
Goals of Coding

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.

For implementing our design into code, we require a high-level functional language. A programming language should have the following characteristics:

Characteristics of Programming Language

Following are the 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.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

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:

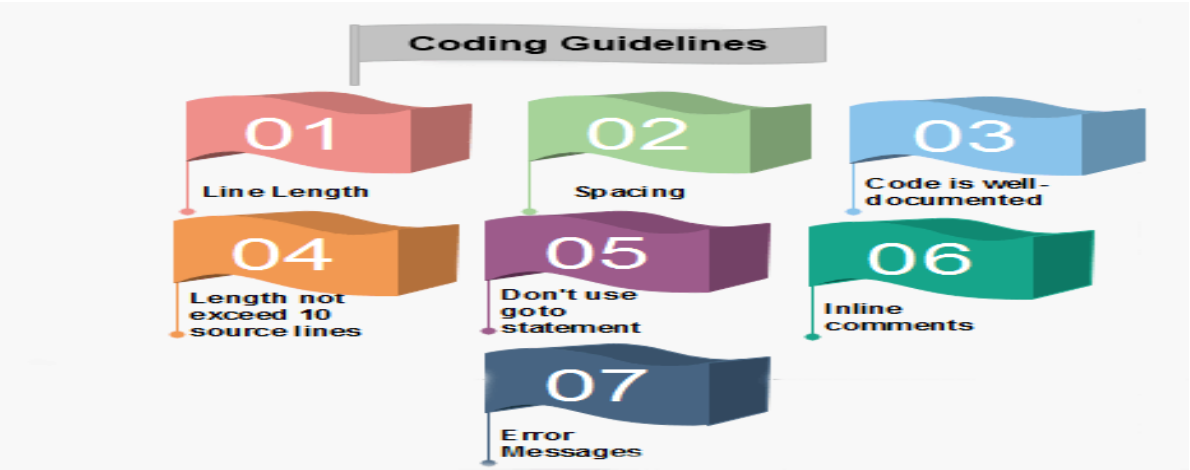
1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs. Indentation should be used to:
   - Emphasize the body of a control structure such as a loop or a select statement.
   - Emphasize the body of a conditional statement
   - Emphasize 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.

DOCUMENTATION 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.

TYPE CHECKING:

- Type checking is the process of verifying and enforcing constraints of types in values.
- A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language.
- It allows the programmer to limit what types may be used in certain circumstances and assigns types to values.
- The type-checker determines whether these values are used appropriately or not.
- It checks the type of objects and reports a type error in the case of a violation, and incorrect types are corrected.
- Whatever the compiler we use, while it is compiling the program, it has to follow the type rules of the language.

- Every language has its own set of type rules for the language. We know that the information about data types is maintained and computed by the compiler.
- The information about data types like INTEGER, FLOAT, CHARACTER, and all the other data types is maintained and computed by the compiler.
- The compiler contains modules, where the type checker is a module of a compiler and its task is type checking.

Conversion

- Conversion from one type to another type is known as **implicit** if it is to be done automatically by the compiler. Implicit type conversions are also called **Coercion** and coercion is limited in many languages.

Example: An integer may be converted to a real but real is not converted to an integer.
Conversion is said to be **Explicit** if the programmer writes something to do the Conversion.
Tasks:
1. has to allow "Indexing is only on an array"
2. has to check the range of data types used
3. INTEGER (int) has a range of -32,768 to +32767
4. FLOAT has a range of 1.2E-38 to 3.4E+38.

Types of Type Checking:

There are two kinds of type checking:

1. Static Type Checking.
2. Dynamic Type Checking.

Static Type Checking:

Static type checking is defined as type checking performed at compile time. It checks the type variables at compile-time, which means the type of the variable is known at the compile time. It generally examines the program text during the translation of the program. Using the type rules of a system, a compiler can infer from the source text that a function (fun) will be applied to an operand (a) of the right type each time the expression fun(a) is evaluated.
Examples of Static checks include:
- **Type-checks:** A compiler should report an error if an operator is applied to an incompatible operand. For example, if an array variable and function variable are added together.
- **The flow of control checks:** Statements that cause the flow of control to leave a construct must have someplace to which to transfer the flow of control. For

example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement, an error occurs if such an enclosing statement does not exist.

- **Uniqueness checks:** There are situations in which an object must be defined only once. For example, in Pascal an identifier must be declared uniquely, labels in a case statement must be distinct, and else a statement in a scaler type may not be represented.
- **Name-related checks:** Sometimes the same name may appear two or more times. For example in Ada, a loop may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

The Benefits of Static Type Checking:

1. Runtime Error Protection.
2. It catches syntactic errors like spurious words or extra punctuation.
3. It catches wrong names like Math and Predefined Naming.
4. Detects incorrect argument types.
5. It catches the wrong number of arguments.
6. It catches wrong return types, like return “70”, from a function that’s declared to return an int.

Dynamic Type Checking:

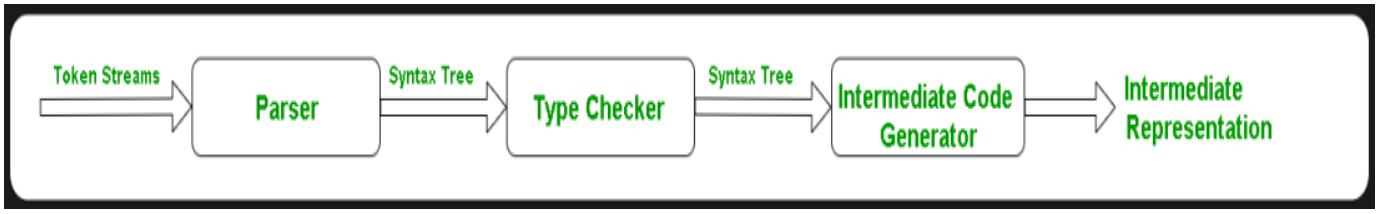
Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed. Dynamic typing results in more compact programs since it is more flexible and does not require types to be spelled out. Programming with a static type system often requires more design and implementation effort.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types, whether it is syntactically correct or not before their execution. Static Type-Checking is also used to determine the amount of memory needed to store the variable.

The design of the type-checker depends on:

1. Syntactic Structure of language constructs.
2. The Expressions of languages.
3. The rules for assigning types to constructs (semantic rules).

The Position of the Type checker in the Compiler:



Type checking in Compiler

The token streams from the lexical analyzer are passed to the PARSER. The PARSER will generate a syntax tree. When a program (source code) is converted into a syntax tree, the type-checker plays a Crucial Role. So, by seeing the syntax tree, you can tell whether each data type is handling the correct variable or not. The Type-Checker will check and if any modifications are present, then it will modify. It produces a syntax tree, and after that, INTERMEDIATE CODE Generation is done.

Overloading:

An Overloading symbol is one that has different operations depending on its context.

Overloading is of two types:

1. Operator Overloading
2. Function Overloading

Operator Overloading: In Mathematics, the arithmetic expression "x+y" has the addition operator '+' is overloaded because '+' in "x+y" have different operators when 'x' and 'y' are integers, complex numbers, reals, and Matrices.

Example: In Ada, the parentheses '()' are overloaded, the ith element of the expression A(i) of an Array A has a different meaning such as a 'call to function 'A' with argument 'i' or an explicit conversion of expression i to type 'A'. In most languages the arithmetic operators are overloaded.

Function Overloading: The Type Checker resolves the Function Overloading based on types of arguments and Numbers.

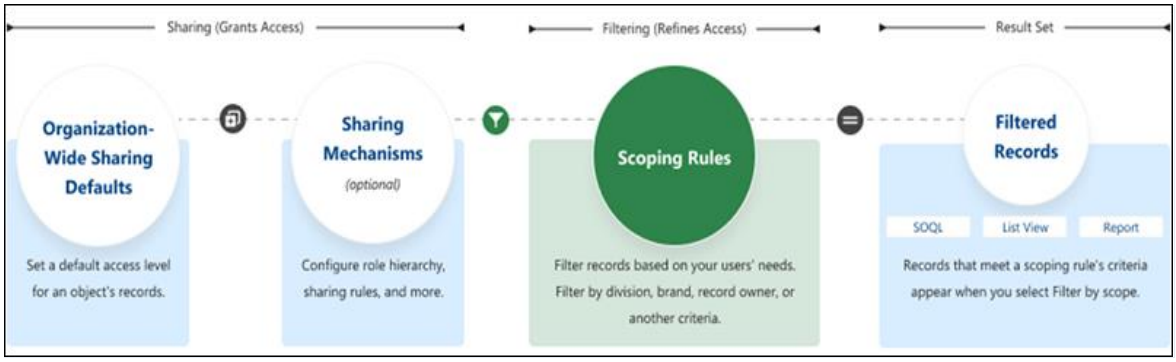
Example:

```
E-->E1(E2)
                {
                   E.type:= if E2.type = s
                   E1.type = s -->t then t
                                  else type_error
                }
```

SCOPING RULES:

Scoping Rules:

- Scoping rules let you control the records that your users see based on criteria that you select.
- You can set up scoping rules for different users in your Salesforce org so that they can focus on the records that matter to them.
- Scoping rules are available for custom objects and the account, case, contact, event, lead, opportunity, and task standard objects. Create, edit, or delete scoping rules using the Tooling or Metadata API, or in Salesforce Setup.



When Do I Use Scoping Rules?

Use scoping rules when you want to let users control the record set that they see. A scoping rule doesn't restrict users' access to other records that they sometimes need. Instead, scoping rules let your users focus on one set of records, then change their focus or search to find a record that's not in the scoped record set when they need to.

For example, you have users who support multiple agencies in your org. Each user is assigned to a specific agency. You can set up scoping rules so that they filter the records that your users see in list views and reports. Users don't have to spend time looking for the correct records, but they still have access to the other agencies' records if they need them.

You can also use scoping rules with Flow Builder to set scope according to a choice your user makes. For example, you have users who work on account records that belong to different divisions in your organization. You want to scope the account records that users see by division, giving your users an easy way to switch between different divisions' record sets. You can set up a flow that your users access using the Lightning Utility Bar to set the scope of records that the user sees in list views, reports, and other features.

How Do Scoping Rules Affect User Access?

Scoping rules are flexible. You can enable and disable them on a query-by-query basis. Plus, they don't restrict the access that your users have to records. Your users can still open and report on all the records that they can access according to your org's sharing settings.

Where Are Scoping Rules Applied?

This table shows how scoping rules work with other Salesforce features.

Feature	Description
List Views	Applied in Lightning Experience if **Filter by scope** is selected
Reports	Applied in Lightning Experience if **Filter by scope** is selected
SOQL	Applied, unless a scope other than *scopingrule* is specified

How Do I Configure Scoping Rules?

Create and manage scoping rules by navigating to a supported object in the Object Manager. Or use the RestrictionRule Tooling API object or RestrictionRule Metadata API type.

Note

The RestrictionRule API object is also used to manage restriction rules. For information on restriction rules, see the Restriction Rule Developer Guide.

When creating more than one scoping or restriction rule, configure the rules so that only one active rule applies to a given user. Salesforce doesn't validate that only one active rule applies for a given user. If you create two active rules, and both rules apply to a given user, only one of the active rules is observed.

After creating rules, you can use a change set or unlocked package to move scoping rules from one org to another.

CONCURRENCY MECHANISM:

What is Software Testing?

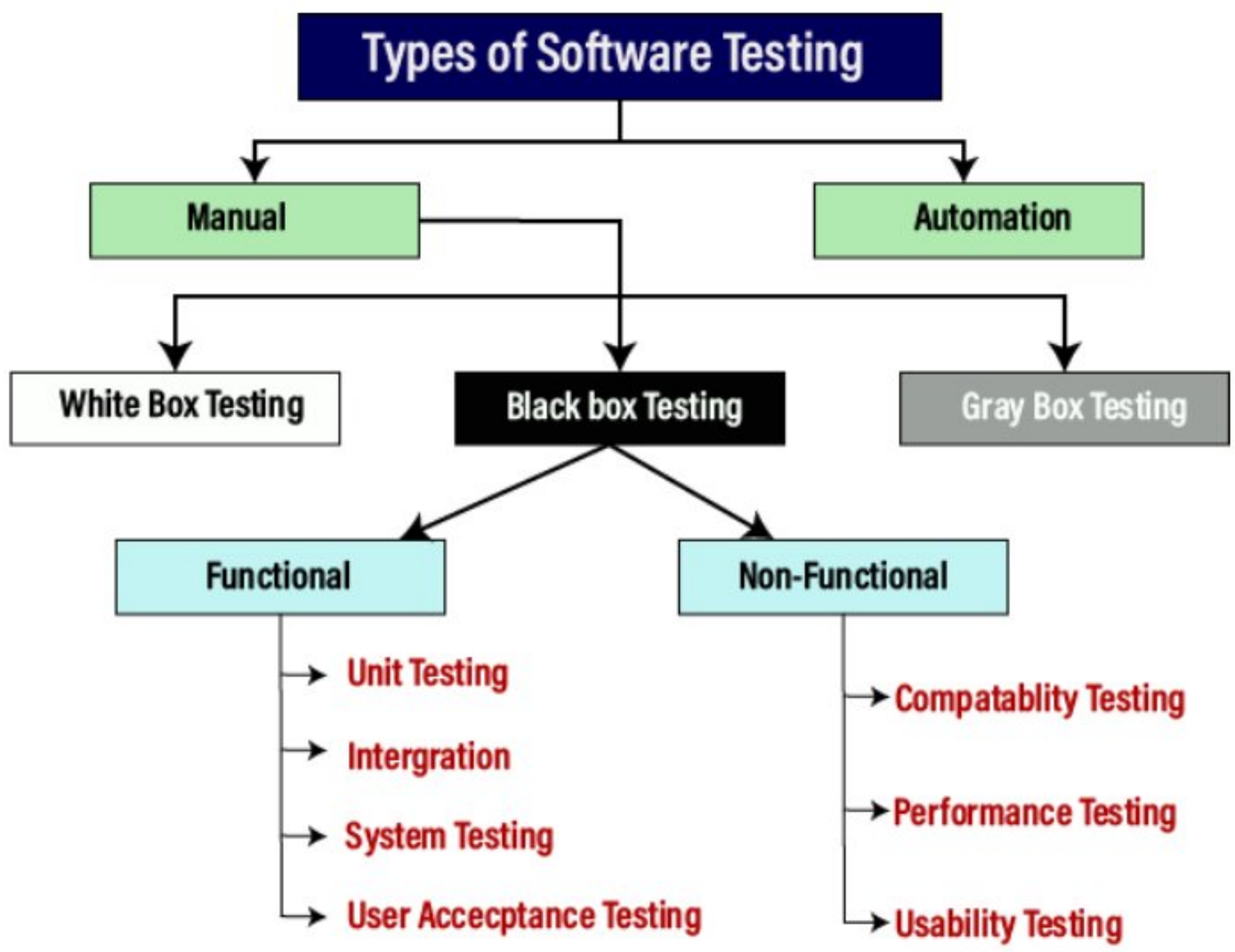
- It is the process of finding the yet undiscovered bugs/errors in the program. The user tests the program, find the error and bugs which could hamper the normal functioning of the program.
- Sometimes, there are changes made to the software, new functionality, or changes to the existing functionalities that are done introduces the possibility of error. It also increases the risk that software would not work as per requirement or does not fulfill the intended purpose.
- Testing reduces this risk. It is through the process of testing and rectification, that we can identify the issues and resolve them.

Why Software Testing is Important?

- Software Testing is Important because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

Types of Software testing

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



Manual testing

- The process of checking the functionality of an application as per the customer needs without taking any help of automation tools is known as manual testing. While performing the manual testing on any application, we do not need any specific knowledge of any testing tool, rather than have a proper understanding of the product so we can easily prepare the test document.

Scanned with OKEN Scanner

Manual testing can be further divided into three types of testing, which are as follows:

- **White box testing**
- **Black box testing**
- **Gray box testing**

Automation testing

- Automation testing is a process of converting any manual test cases into the test scripts with the help of automation tools, or any programming language is known as automation testing. With the help of automation testing, we can enhance the speed of our test execution because here, we do not require any human efforts. We need to write a test script and execute those scripts.

What are the benefits of Software Testing?

Here are the benefits of using software testing:

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

Testing Strategies:

- Software is tested to uncover errors introduced during design and construction. Testing often accounts for More project effort than other s/e activity. Hence it has to be done carefully using a testing strategy.
- The strategy is developed by the project manager, software engineers and testing specialists.
- Testing is the process of execution of a program with the intention of finding errors Involves 40% of total project cost Testing Strategy provides a road map that describes the steps to be conducted as part of testing.
- It should incorporate test planning, test case design, test execution and resultant data collection and execution Validation refers to a different set of activities that ensures that the software is traceable to theCustomer requirements.
- V&V encompasses a wide array of Software Quality Assurance

A strategic Approach for Software testing

- Testing is a set of activities that can be planned in advance and conducted systematically. Testing strategy Should have the following characteristics:
  - usage of Formal Technical reviews(FTR)
  - Begins at component level and covers entire system

Scanned with OKEN Scanner

- Different techniques at different points
- conducted by developer and test group
- should include debugging

➢ Software testing is one element of verification and validation.

Verification refers to the set of activities that ensure that software correctly implements a specific function. (Ex: Are we building the product right?)

Validation refers to the set of activities that ensure that the software built is traceable to customer requirements. (Ex: Are we building the right product ?)

Types of Testing Techniques / Terminologies

The testers should possess good knowledge of the software or application. There should not any critical bug or error should be left in the application.

➢ Testing can be done by software developer and independent testing group. Testing and debugging are different activities. Debugging follows testing Low level tests verifies small code segments. High level tests validate major system functions against customer requirements

➢ Test Strategies for Conventional Software:

➢ Testing Strategies for Conventional Software can be viewed as a spiral consisting of four levels of testing:

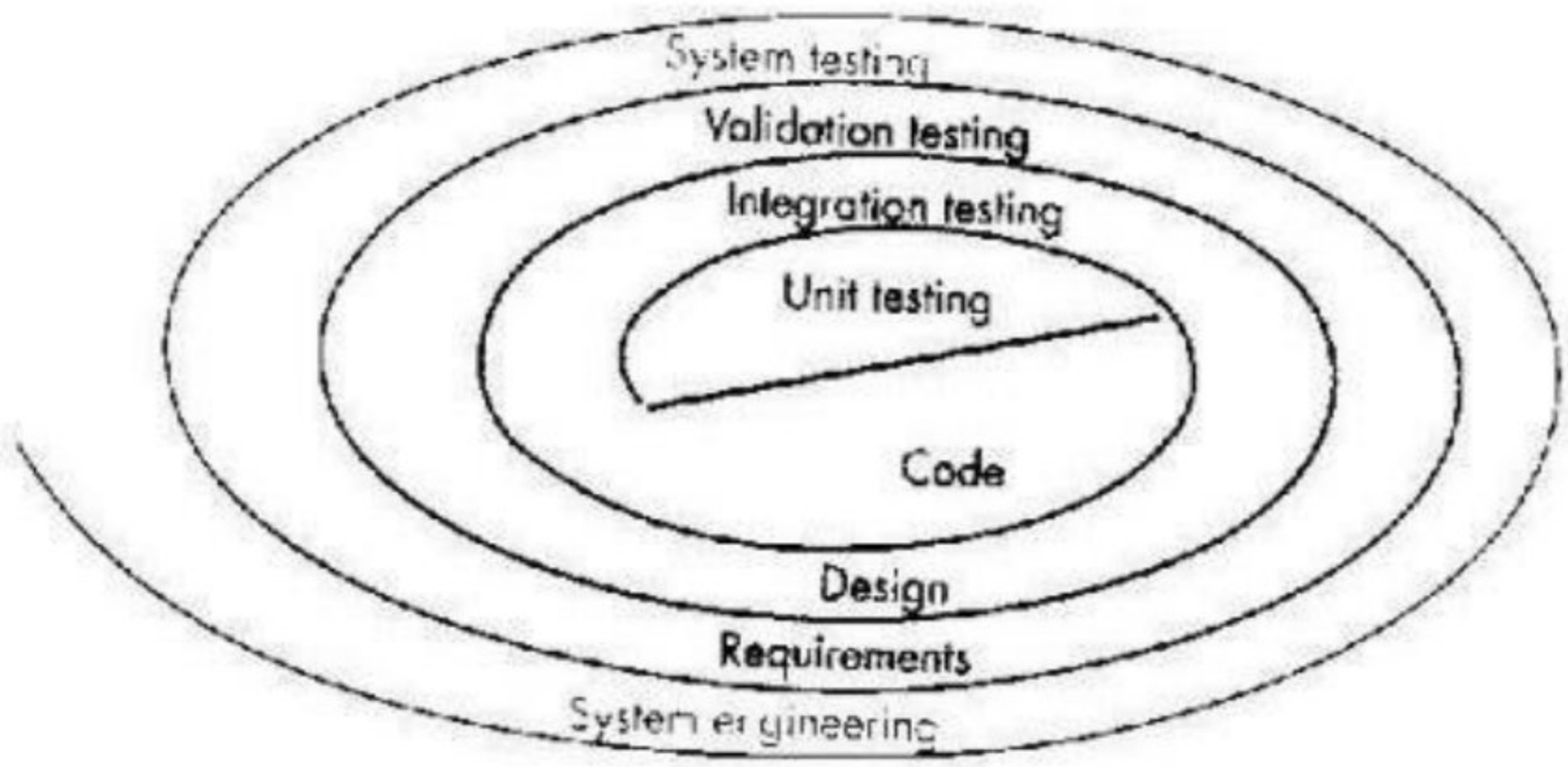
1) Unit Testing

2)Integration Testing

3)Validation Testing and

4) System Testing

Spiral Representation of Testing for Conventional Software



Scanned with OKEN Scanner

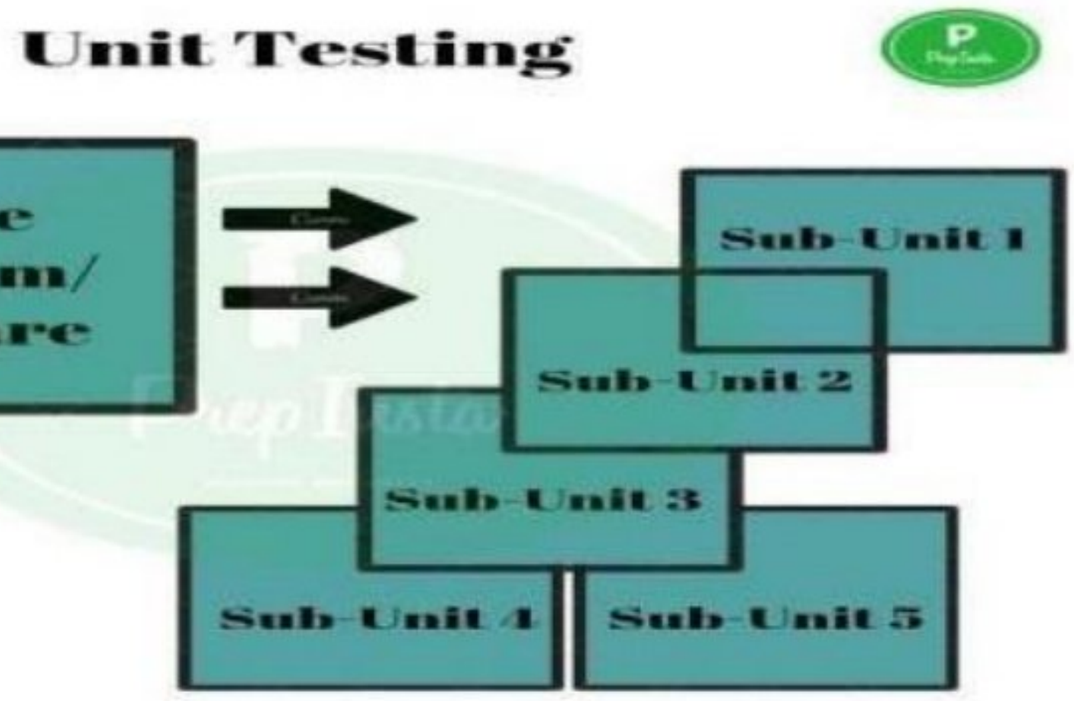


Fig: Unit Testing

- **Unit Testing** – It refers to individual units or components of the software that are tested. The purpose of unit testing is to see that each software unit performs as per the expected requirement and functionality. It may be termed as first-level testing.
- **Integration Testing** - It refers to testing in which the individual software components or modules are combined together and tested as a group. It is used to see whether there is any fault in the integrated units. It is normally performed after the unit testing.
- **System Testing –** The testing is performed on the system as a whole and to see whether the system is working as per the requirements and functionality specified. It takes the components of the integration testing as input and performs the testing.
- **Regression Testing –** This testing is performed to check whether the changes to the software do not produce undesirable results. It is used to ensure that the previously developed software components and tested components still perform correctly after the changes. It also depicts that the changes have not broken any functionality.
- **UAT – User Acceptance testing –** It is the testing done by the end-user or client to accept the software system before moving the software to the production environment. It is used to check the functionality of the software end to end related to the business flow.

The testers should possess good knowledge of the software or application. There should not any critical bug or error should be left in the application.

- **Black Box Testing** – In this type of testing, the emphasis is on the functionality of the application. The code for the tester is like a black box. He concentrates on the functionality rather than the internal structures of the application. It is used to check the application from the user's perspective and per requirements stated by the customer.
- **White Box Testing** - In this type of testing, the emphasis is given on the internal structure of the program in the application/software, code structure, internal design to verify the input and output of the process in the software.
- **Grey Box Testing** – This software testing method is a combination of Black Box testing and White Box Testing. It includes the testing of internal coding structure so involves the inputs from both developer and tester to ensure a quality product.
- **Ad-hoc Testing –** It is also one of the terms used in Software Testing. The testing is performed without planning and other formal documentation. It is normally intended to run once. This testing is carried out on a random basis to check the functionality of the application.

Scanned with OKEN Scanner

- **Exploratory Testing –** In this testing, the tester explores the application what intends to do, or its purpose. The tester uses his experience and knowledge to check the application and try to find errors/bugs. We can say it learning along with the test execution.
- **Alpha Testing –** This testing is performed before the final product is released to the end-users or the customer. This testing is basically performed by the QA Testers internally to the organization or you can say performed at the developer's site. The Bugs can be addressed by the developer immediately and can be fixed.
- **Beta Testing –** This testing is performed by the end-users or at the customer's place who are not part of the organization. It is normally performed at the client's place. The issues collected from the testing will be implemented for future versions of the product or application. The user inputs are gathered on the product and ensure it is ready for the customers.
- **Automated Testing -** In this testing, the Test cases execution takes using an Automation tool. The Test data is prepared, test cases are executed using the required data. Test scripts are created for the testing of the application.

FUNCTIONAL TESTING:

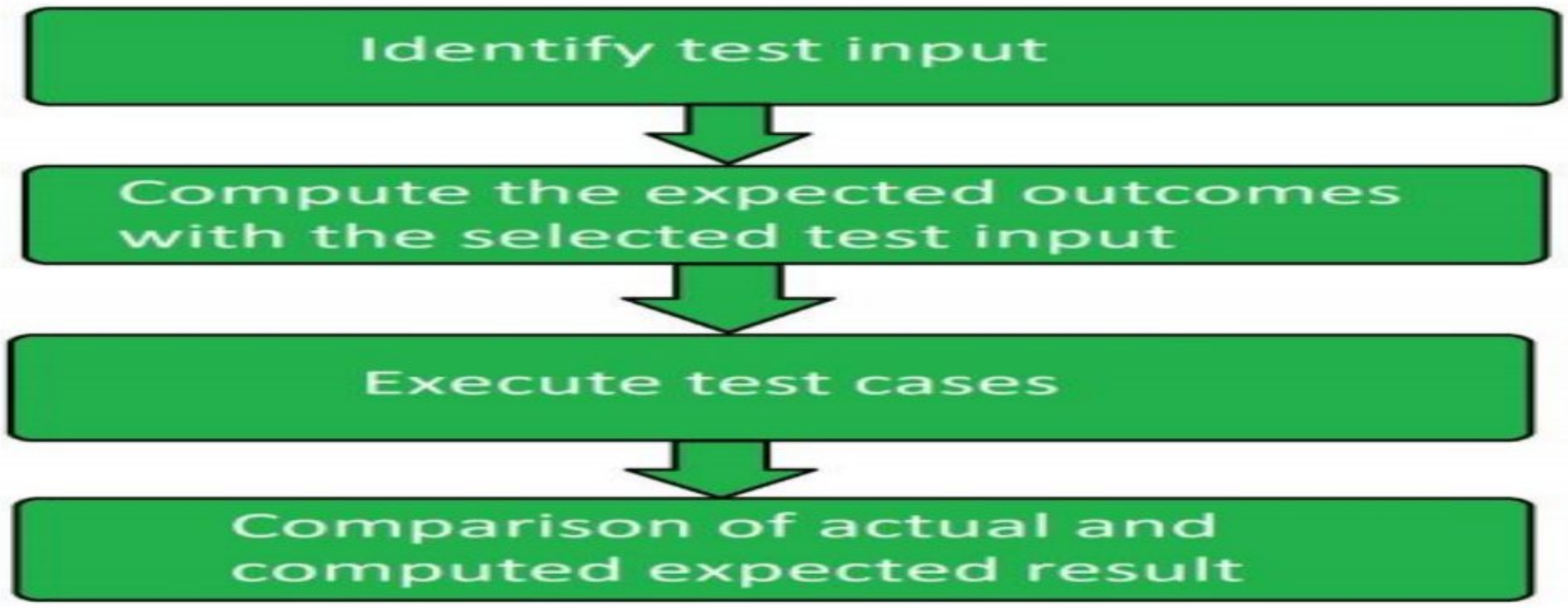
- Functional Testing is a type of Software Testing in which the system is tested against the functional requirements and specifications.
- Functional testing ensures that the requirements or specifications are properly satisfied by the application.
- This type of testing is particularly concerned with the result of processing.
- It focuses on simulation of actual system usage but does not develop any system structure assumptions.
- It is basically defined as a type of testing which verifies that each function of the software application works in conformance with the requirement and specification.
- This testing is not concerned about the source code of the application.
- Each functionality of the software application is tested by providing appropriate test input, expecting the output and comparing the actual output with the expected output.
- This testing focuses on checking of user interface, APIs, database, security, client or server application and functionality of the Application Under Test.
- Functional testing can be manual or automated.

Functional Testing Process:

Functional testing involves the following steps:

i. Identify function that is to be performed.
ii. Create input data based on the specifications of function.
iii. Determine the output based on the specifications of function.
iv. Execute the test case.
v. Compare the actual and expected output.

Scanned with OKEN Scanner



Major Functional Testing Techniques:

- Unit Testing
- Integration Testing
- Smoke Testing
- User Acceptance Testing
- Interface Testing
- Usability Testing
- System Testing
- Regression Testing

Functional Testing Tools:

- Sele Selenium
- QTP
- JUnit
- SoapUI

Advantages of Functional Testing:

- It ensures to deliver a bug-free product.
- It ensures to deliver a high-quality product.
- No assumptions about the structure of the system.
- This testing is focused on the specifications as per the customer usage.

Disadvantages of Functional Testing:

- There are high chances of performing redundant testing.
- Logical errors can be missed out in the product.
- If the requirement is not complete then performing this testing becomes difficult.

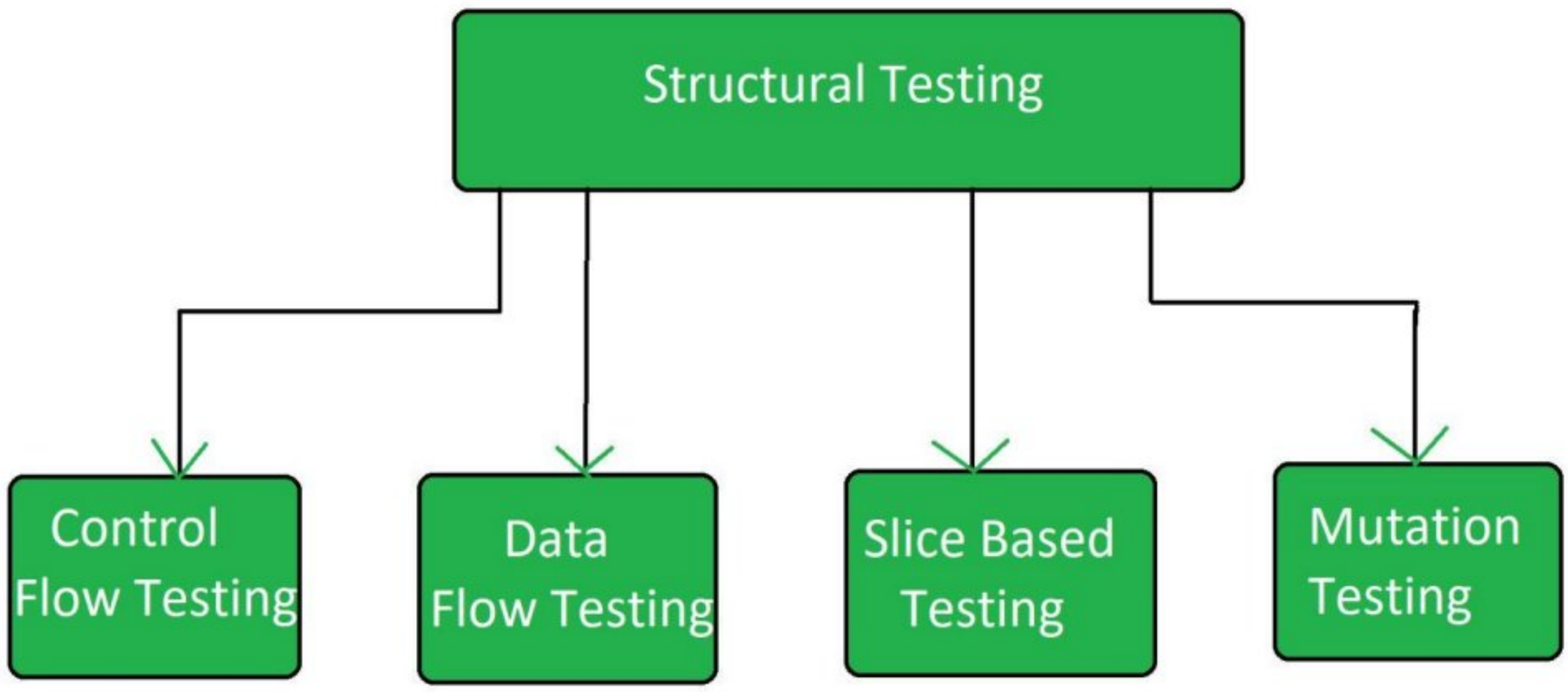
STRUCTURAL TESTING:

Structural testing is a type of software testing which uses the internal design of the software for testing or in other words the software testing which is performed by the team which knows the development phase of the software, is known as structural testing. Structural testing is basically related to the internal design and implementation of the software i.e. it involves the development team members in the testing team. It basically tests different aspects of the software according to its types. Structural testing is just the opposite of behavioral testing.

Scanned with OKEN Scanner

Types of Structural Testing:

There are 4 types of Structural Testing:



- **Control Flow Testing:**
  Control flow testing is a type of structural testing that uses the programs's control flow as a model. The entire code, design and structure of the software have to be known for this type of testing. Often this type of testing is used by the developers to test their own code and implementation. This method is used to test the logic of the code so that required result can be obtained.
- **Data Flow Testing:**
  It uses the control flow graph to explore the unreasonable things that can happen to data.
  The detection of data flow anomalies are based on the associations between values and variables. Without being initialized usage of variables. Initialized variables are not used once.
- **Slice Based Testing:**
  It was originally proposed by Weiser and Gallagher for the software maintenance. It is useful for software debugging, software maintenance, program understanding and quantification of functional cohesion. It divides the program into different slices and tests that slice which can majorly affect the entire software.
- **Mutation Testing:**
  Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

Advantages of Structural Testing:

- It provides thorough testing of the software.
- It helps in finding out defects at an early stage.
- It helps in elimination of dead code.
- It is not time consuming as it is mostly automated.

Disadvantages of Structural Testing:

- It requires knowledge of the code to perform test.
- It requires training in the tool used for testing.
- Sometimes it is expensive.

Scanned with OKEN Scanner

Structural Testing Tools:

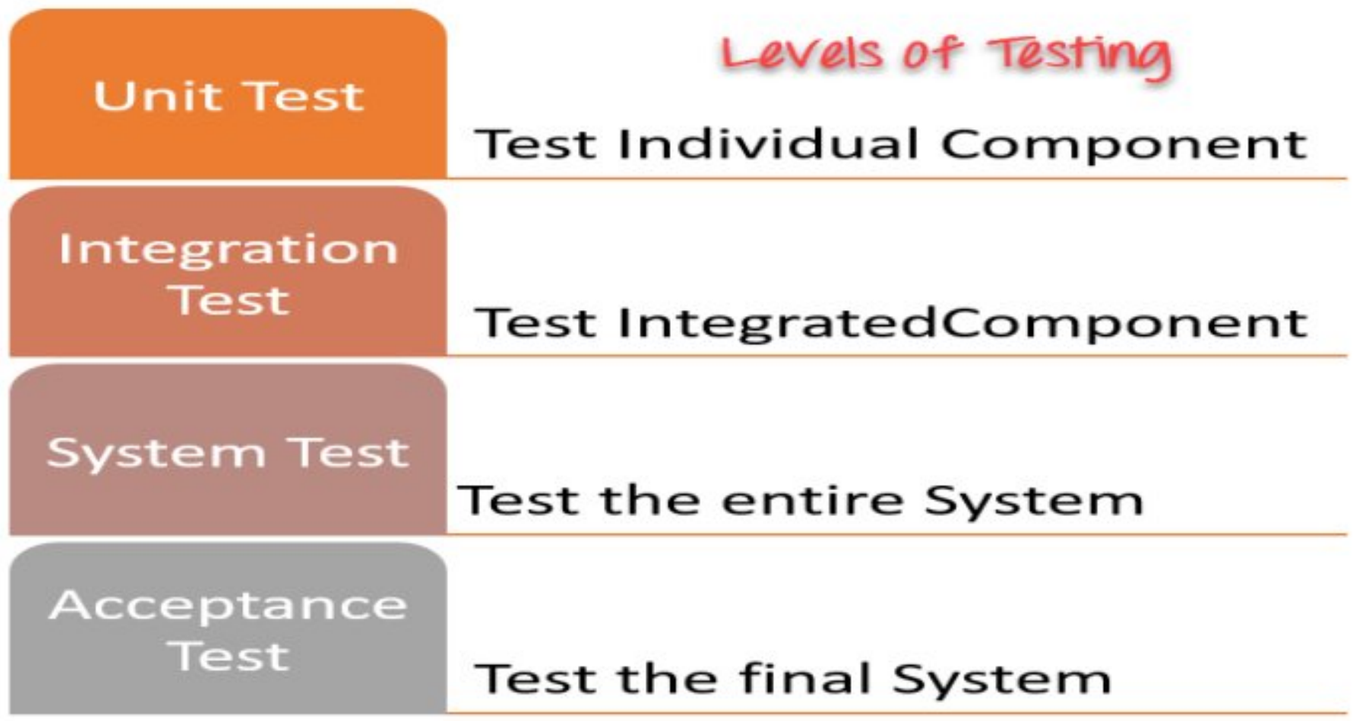
- JBehave
- Cucumber
- Junit
- Cfix

LEVELS OF TESTING:

4 Levels of Testing

There are mainly four **Levels of Testing** in software testing :

1. **Unit Testing** : checks if software components are fulfilling functionalities or not.
2. **Integration Testing** : checks the data flow from one module to other modules.
3. **System Testing** : evaluates both functional and non-functional needs for the testing.
4. **Acceptance Testing** : checks the requirements of a specification or contract are met as per its delivery.



Each Testing Level Details

Unit testing:

- A Unit is a smallest testable portion of system or application which can be compiled, liked, loaded, and executed. This kind of testing helps to test each module separately.
- The aim is to test each part of the software by separating it. It checks that component are fulfilling functionalities or not. This kind of testing is performed by developers.

Scanned with OKEN Scanner

Integration testing:

- Integration means combining. For Example, In this testing phase, different software modules are combined and tested as a group to make sure that integrated system is ready for system testing.
- Integrating testing checks the data flow from one module to other modules. This kind of testing is performed by testers.

System Testing

- System testing is performed on a complete, integrated system. It allows checking system's compliance as per the requirements. It tests the overall interaction of components. It involves load, performance, reliability and security testing.
- System testing most often the final test to verify that the system meets the specification. It evaluates both functional and non-functional need for the testing.

Acceptance testing:

- Acceptance testing is a test conducted to find if the requirements of a specification or contract are met as per its delivery. Acceptance testing is basically done by the user or customer. However, other stockholders can be involved in this process.

VALIDATION TESTING:

What is Validation Testing?

Validation is the process of determining if the completed product fulfills the customer's expectations and criteria. It is a dynamic validation and testing technique for the real product.

The software verification and validation testing is the process after the validation testing stage is secondary to verification testing.

Why is Validation Testing Important?

The Advantages of Validation Testing :

- *To ensure customer satisfaction*
- *To be confident about the product*
- *To fulfill the client's requirement until the optimum capacity*
- *Software acceptance from the end-user*

Scanned with OKEN Scanner

What are the Phases of the Validation Testing Process?

- **Define Requirements**– Mapping out the plan for the requirement gathering process. This will not only include planning out the entire process beforehand but also mapping out the exact requirements that are needed.
- **Team Selection**– To get a qualified, knowledgeable, and skilled team onboard. The team selection process involves selecting the individuals as per their past capacity and technical tuning so that they can easily attune themselves to the nature of the bug.

- **Maintaining Documentation**– Any form of testing requires voluminous user specification documentation along with several release cases, test cases, and manuals that have to be jolted down so that they are no confusion even in case of an exit of any core member of the team.
- **Validation Report**– The software is evaluated as per user specifications and a proper validation report is submitted in order to cross-check the evaluations along with getting an estimated date and round-off for the bug removal and for the system to start functioning properly.
- **Incorporation of changes**–Incorporate the changes that have been validated in the last stage.

What are the Types of Validation Testing?

Validation testing types a V-shaped testing pattern, which includes its variations and all the activities that it consists of are:

Unit Testing – It is an important type of validation testing. The point of the unit testing is to search for bugs in the product segment. Simultaneously, it additionally confirms crafted modules and articles which can be tried independently.

Integration testing -This is a significant piece of the validation model wherein the interaction between, where the association between the various interfaces of the pertaining component is tried. Alongside the communication between the various pieces of the framework, the connection of the framework with the PC working framework, document framework, equipment, and some other programming framework it may cooperate with, is likewise tried.

System testing – System testing is done when the whole programming framework is prepared. The principal worry of framework testing is to confirm the framework against the predefined necessities. While doing the tests, the tester isn't worried about the internals of the framework however checks if the framework acts according to desires.

Scanned with OKEN Scanner

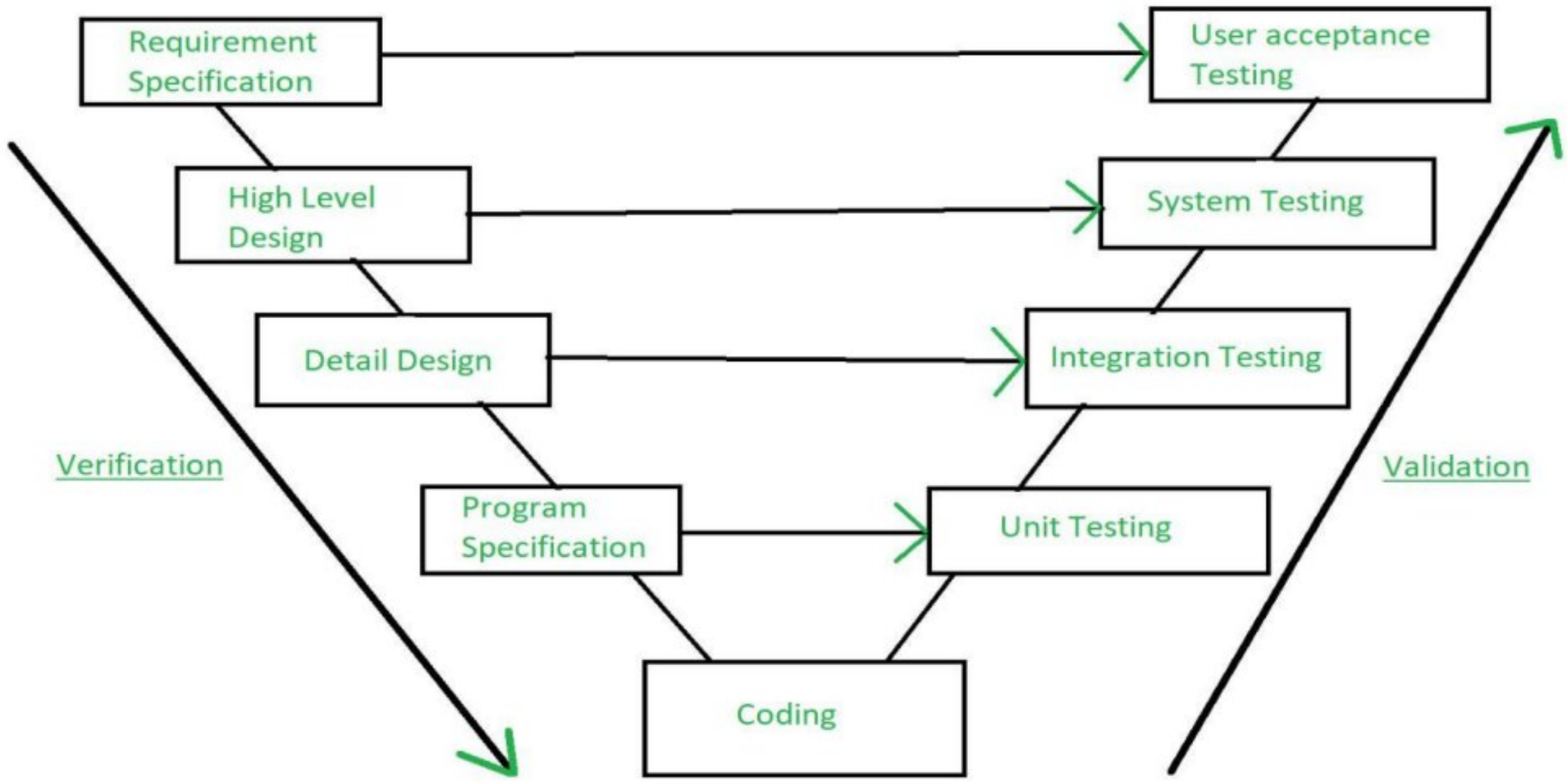
User acceptance testing – During this testing, the tester actually needs to think like the customer and test the product concerning client needs, prerequisites, and business forms and decide if the product can be given over to the customer or not.

Following are the important differences between Verification and Validation

Sr. No.	Key	Verification	Validation
1	Definition	Verification is the process in which product or system is evaluated in development phase to find out whether it meets the specified requirements or not.	On other hand Validation is the process in which product or system is evaluated at the end of the development process to determine whether software meets the customer expectations and requirements or not.
2	Objective	The main objective of Verification process is to make sure that the system being develop is as per the requirements and design specifications of the customer and if it deviates from it then make it correct in the development phase itself.	On other hand the objective of Validation is to make sure that the product which has been developed is actually meet up the user's requirements or not. And if it is not then make it to the level of acceptance in re development phase.
3	Activities	Main activities which defines the Verification process are Reviews of specification and product development, Meetings about diversification and inspections.	On other hand activities under Validation process are typically different type of testing such as Black Box testing, White Box testing, Grey box testing etc. which ensure the defect free delivery of product as per specification document.
4	Type	Verification is the process where execution of code is not take place and hence it comes under static testing.	On other hand during Validation execution of code take place and thus it comes under dynamic testing.
5	Sequence	Verification is carried out before the Validation.	On other hand Validation activity is carried out just after the Verification.
6	Performer	Verification is carried out by	On other hand Validation is

Scanned with OKEN Scanner

Sr. No.	Key	Verification	Validation
		Quality assurance team.	executed on software code with the help of testing team.



THE ART OF DEBUGGING:

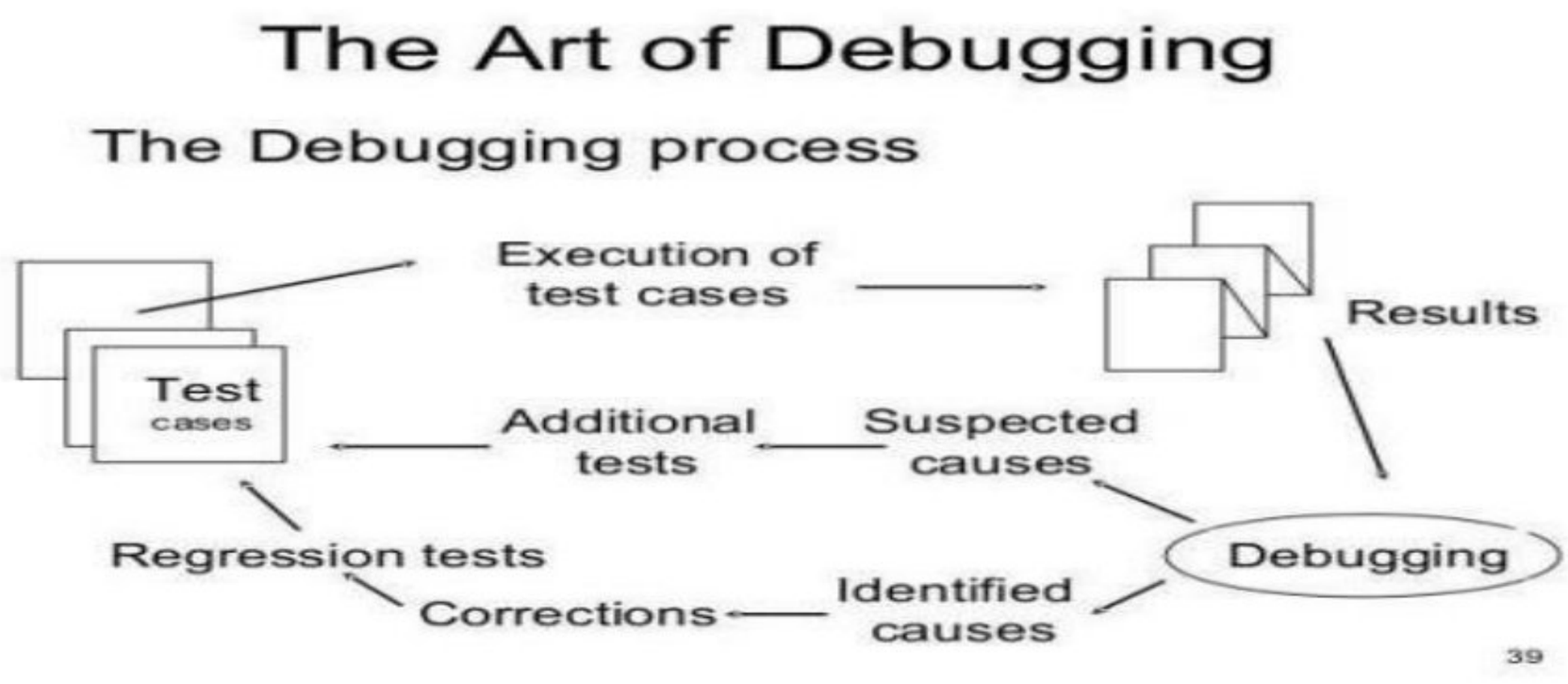


Fig: Debugging process

In the context of software engineering, debugging is the process of fixing a bug in the software. In other words,

it refers to identifying, analyzing, and removing errors. This activity begins after the software fails to execute properly

and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex

and tedious task because errors need to be resolved at all stages of debugging.

Scanned with OKEN Scanner

Debugging Process:

Steps involved in debugging are:

I. Problem identification and report preparation.
II. Assigning the report to the software engineer to the defect to verify that it is genuine.
III. Defect Analysis using modeling, documentation, finding and testing candidate flaws, etc.
IV. Defect Resolution by making required changes to the system.
V. Validation of corrections.

The debugging process will always have one of two outcomes :

1. The cause will be found and corrected.

2. The cause will not be found.

- Later, the person performing debugging may suspect a cause, design a test case to help validate that suspicion and work toward error correction in an iterative fashion.
- During debugging, we encounter errors that range from mildly annoying to catastrophic. As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches/Strategies:

1.Brute Force: Study the system for a larger duration in order to understand the system. It helps the debugger to construct different representations of systems to be debugging depending on the need. A study of the system is also done actively to find recent changes made to the software.

2. Backtracking: Backward analysis of the problem which involves tracing the program backward from the location of the failure message in order to identify the region of faulty code. A detailed study of the region is conducted to find the cause of defects.

3.Forward analysis of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused on to find the defect.

4.Using the past experience of the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.

5.Cause elimination: it introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.

Debugging Tools:

Debugging tool is a computer program that is used to test and debug other programs. A lot of public domain software like gdb and dbx are available for debugging. They offer console-based command-line interfaces. Examples of automated debugging tools include code based tracers, profilers, interpreters, etc. Some of the widely used debuggers are:

1. Radare2
2. WinDbg
3. Valgrind

Difference Between Debugging and Testing:

Debugging is different from testing. Testing focuses on finding bugs, errors, etc whereas debugging starts after a bug has been identified in the software. Testing is used to ensure that

Scanned with OKEN Scanner

the program is correct and it was supposed to do with a certain minimum success rate. Testing can be manual or automated. There are several different types of testing like unit testing, integration testing, alpha and beta testing, etc. Debugging requires a lot of knowledge, skills, and expertise. It can be supported by some automated tools available but is more of a manual process as every bug is different and requires a different technique, unlike a pre-defined testing mechanism.

TESTING TOOLS:

- Software testing tools are required for the betterment of the application or software.
- That's why we have so many tools available in the market where some are open-source and paid tools.
- The significant difference between open-source and the paid tool is that the open-source tools have limited features, whereas paid tool or commercial tools have no limitation for the features. The selection of tools depends on the user's requirements, whether it is paid or free.
- The software testing tools can be categorized, depending on the licensing (paid or commercial, open-source), technology usage, type of testing, and so on.

With the help of testing tools, we can improve our software performance, deliver a high-quality product, and reduce the duration of testing, which is spent on manual efforts.

The software testing tools can be divided into the following:

- **Test management tool**
- **Bug tracking tool**
- **Automated testing tool**
- **Performance testing tool**
- **Cross-browser testing tool**
- **Integration testing tool**
- **Unit testing tool**
- **Mobile/android testing tool**
- **GUI testing tool**
- **Security testing tool**

Scanned with OKEN Scanner

Test management tool

Test management tools are used to keep track of all the testing activity, fast data analysis, manage manual and automation test cases, various environments, and plan and maintain manual testing as well.

Bug tracking tool

The defect tracking tool is used to keep track of the bug fixes and ensure the delivery of a quality product. This tool can help us to find the bugs in the testing stage so that we can get the defect-free data in the production server. With the help of these tools, the end-users can allow reporting the bugs and issues directly on their applications.

Automation testing tool

This type of tool is used to enhance the productivity of the product and improve the accuracy. We can reduce the time and cost of the application by writing some test scripts in any programming language.

Performance testing tool

Performance or Load testing tools are used to check the load, stability, and scalability of the application. When n-number of the users using the application at the same time, and if the application gets crashed because of the immense load, to get through this type of issue, we need load testing tools.

Cross-browser testing tool

This type of tool is used when we need to compare a web application in the various web browser platforms. It is an important part when we are developing a project. With the help of these tools, we will ensure the consistent behavior of the application in multiple devices, browsers, and platforms.

Integration testing tool

This type of tool is used to test the interface between modules and find the critical bugs that are happened because of the different modules and ensuring that all the modules are working as per the client requirements.

Unit testing tool

This testing tool is used to help the programmers to improve their code quality, and with the help of these tools, they can reduce the time of code and the overall cost of the software.

Scanned with OKEN Scanner

Mobile/android testing tool

We can use this type of tool when we are testing any mobile application. Some of the tools are open-source, and some of the tools are licensed. Each tool has its functionality and features.

GUI testing tool

GUI testing tool is used to test the User interface of the application because a proper GUI (graphical user interface) is always useful to grab the user's attention. These type of tools will help to find the loopholes in the application's design and makes its better.

Security testing tool

The security testing tool is used to ensure the security of the software and check for the security leakage. If any security loophole is there, it could be fixed at the early stage of the product. We need this type of the tool when the software has encoded the security code which is not accessible by the unauthorized users.

Scanned with OKEN Scanner