MOHAN BABU UNIVERSITY
Sree Sainath Nagar, Tirupati 517 102

Module 3 DESIGN ENGINEERING AND METRICS

Design Engineering: *Design within the context of Software Engineering, The Design process, Design concepts, Software Architecture, Architectural Styles, Architectural design, Pattern based Design-Design Patterns, Pattern based software design.*
Process and Project Metrics: Metrics in the process and project domains, Software Measurement, Metrics for software quality.

Design Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design. The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed. The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model. The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.The importance of software design can be stated with a single word—quality. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder‟s requirements into a finished software product or system.

The design process

Software design is an iterative process through which requirements are translated into a ―blueprint‖ for constructing the software. Initially, the blueprint depicts a

holistic view of software. That is, the design is represented at a high level of abstraction— a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle

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:
Interface Design
Architectural Design
Detailed Design

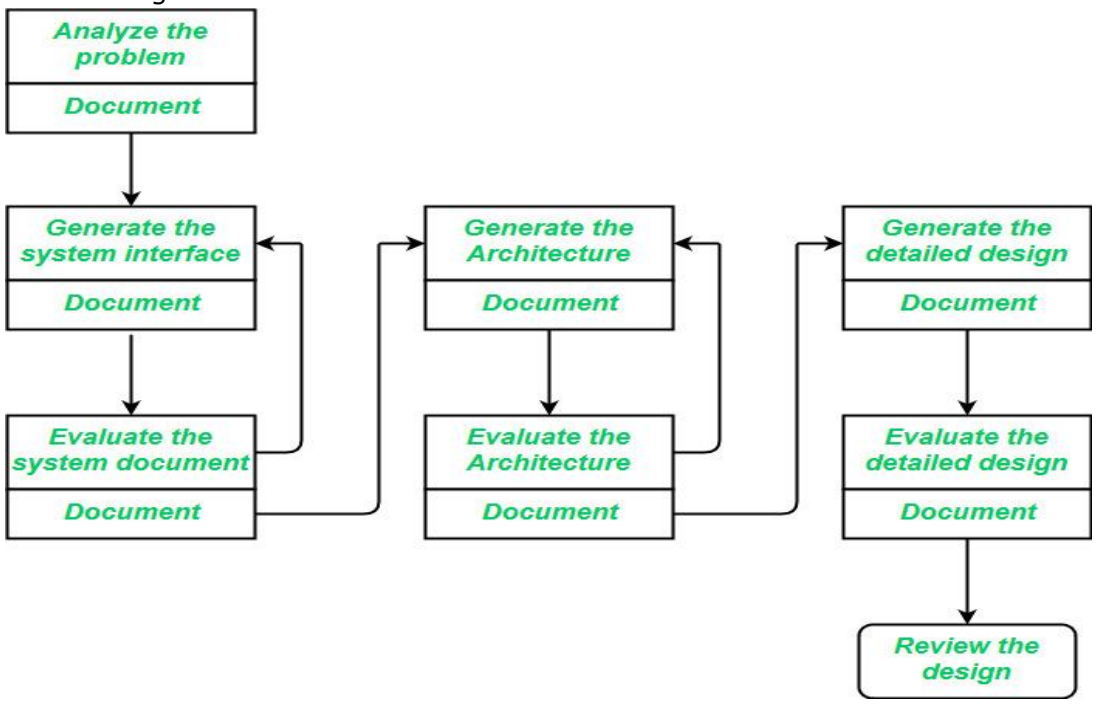


Fig: Design Process

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.

Issues in architectural design includes:

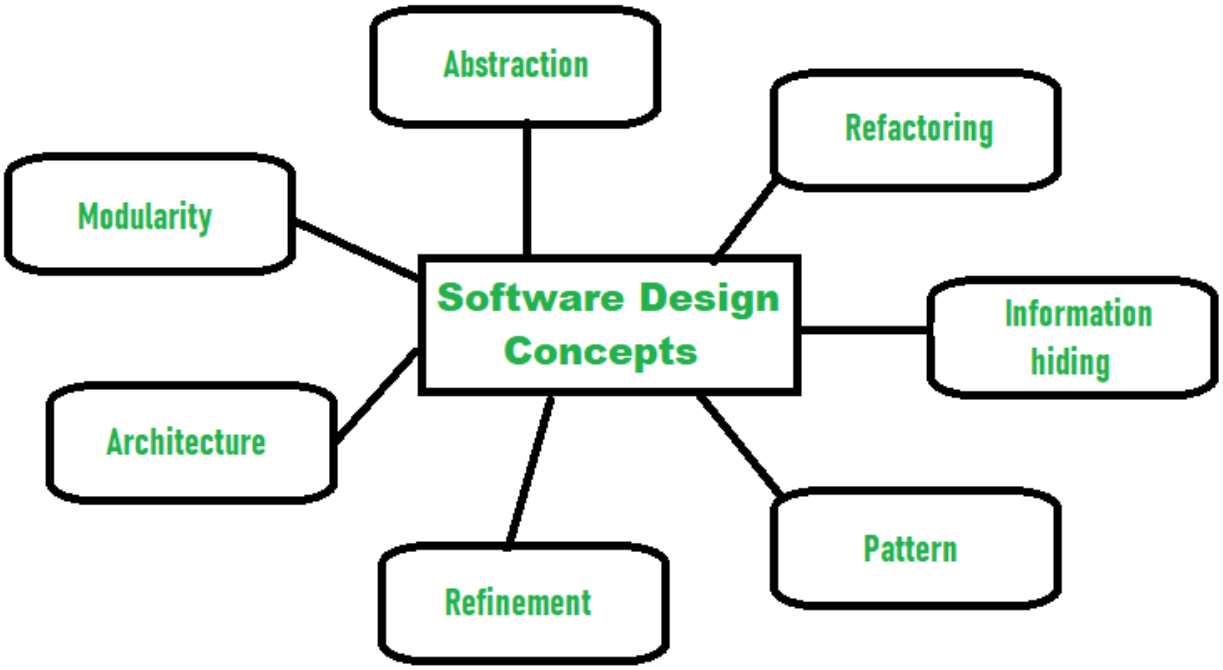
- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between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

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:

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.

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.

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.

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.

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.

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.

Refactoring- reconstruct something

Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".

Software architecture

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.

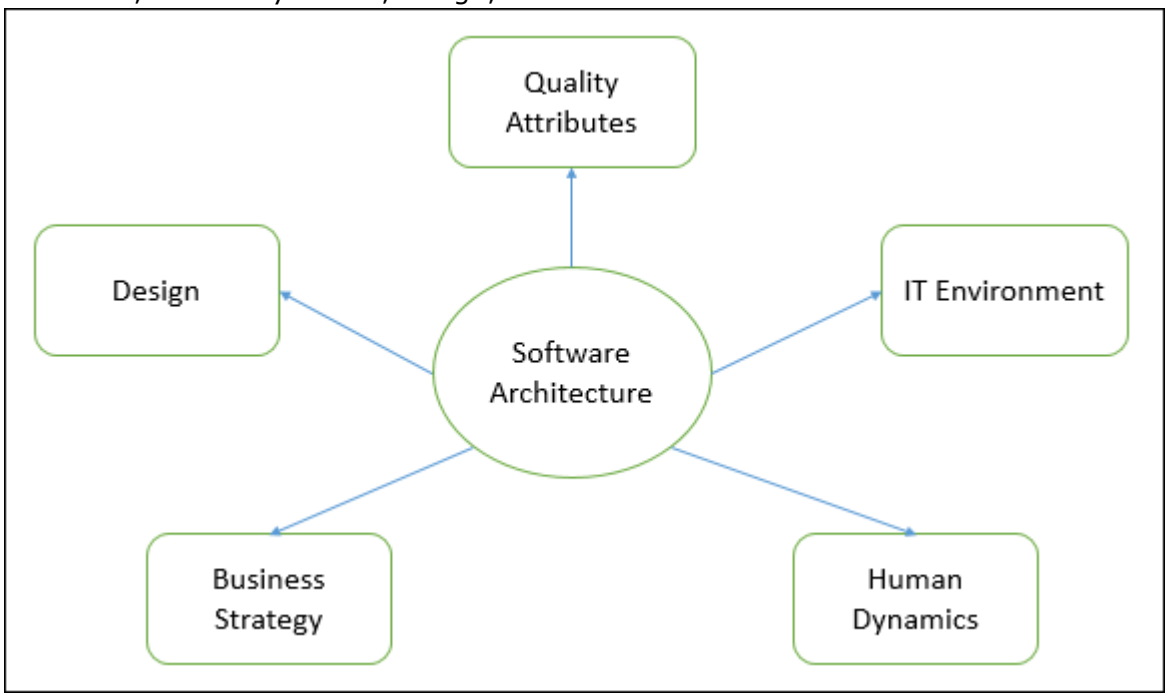


Fig: Software architecture

Architecture serves as a blueprint for a system. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

It defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of

- Selection of structural elements and their interfaces by which the system is composed.
- Behavior as specified in collaborations among those elements.
- Composition of these structural and behavioral elements into large subsystem.
- Architectural decisions align with business objectives.
- Architectural styles guide the organization.

  Goals of Architecture

  The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.

  Some of the other goals are as follows –

  Expose the structure of the system, but hide its implementation details.

  Realize all the use-cases and scenarios.

  Try to address the requirements of various stakeholders.

  Handle both functional and quality requirements.

  Reduce the goal of ownership and improve the organization’s market position.

  Improve quality and functionality offered by the system.

  Improve external confidence in either the organization or system.

  Limitations

  Software architecture is still an emerging discipline within software engineering. It has the following limitations –

  Lack of tools and standardized ways to represent architecture.

  Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.

  Lack of awareness of the importance of architectural design to software development.

Lack of understanding of the role of software architect and poor communication among stakeholders.

Lack of understanding of the design process, design experience and evaluation of design.

Role of Software Architect
A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas –

Design Expertise
Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.

Lead the development team and coordinate the development efforts for the integrity of the design.

Should be able to review design proposals and tradeoff among themselves.

Domain Expertise
Expert on the system being developed and plan for software evolution.

Assist in the requirement investigation process, assuring completeness and consistency.

Coordinate the definition of domain model for the system being developed.

Technology Expertise
Expert on available technologies that helps in the implementation of the system.

Coordinate the selection of programming language, framework, platforms, databases, etc.

Methodological Expertise
Expert on software development methodologies that may be adopted during SDLC (Software Development Life Cycle).

Choose the appropriate approaches for development that helps the entire team.

Hidden Role of Software Architect
Facilitates the technical work among team members and reinforcing the trust relationship in the team.

Information specialist who shares knowledge and has vast experience.

Protect the team members from external forces that would distract them and bring less value to the project.

Deliverables of the Architect
A clear, complete, consistent, and achievable set of functional goals

A functional description of the system, with at least two layers of decomposition

A concept for the system

A design in the form of the system, with at least two layers of decomposition

A notion of the timing, operator attributes, and the implementation and operation plans

A document or process which ensures functional decomposition is followed, and the form of interfaces is controlled.

Architectural styles

The software needs the architectural design to represents the design of software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles.
Each style will describe a system category that consists of :

A set of components(eg: a database, computational modules) that will perform a function required by the system.
The set of connectors will help in coordination, communication, and cooperation between the components.
Conditions that how components can be integrated to form the system.
Semantic models that help the designer to understand the overall properties of the system.
The use of architectural styles is to establish a structure for all the components of the system.

Taxonomy of Architectural styles:

Data centered architectures:
A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
The figure illustrates a typical data centered style. The client software access a central repository. Variation of this approach are used to transform the repository

into a blackboard when data related to client or data of interest for the client change the notifications to client software.

This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.

Data can be passed among clients using blackboard mechanism.

Data flow architectures:

This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.

The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.

Pipes are used to transmit data from one component to the next.

Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.

If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

Call and Return architectures: It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

Remote procedure call architecture: This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.

Main program or Subprogram architectures: The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

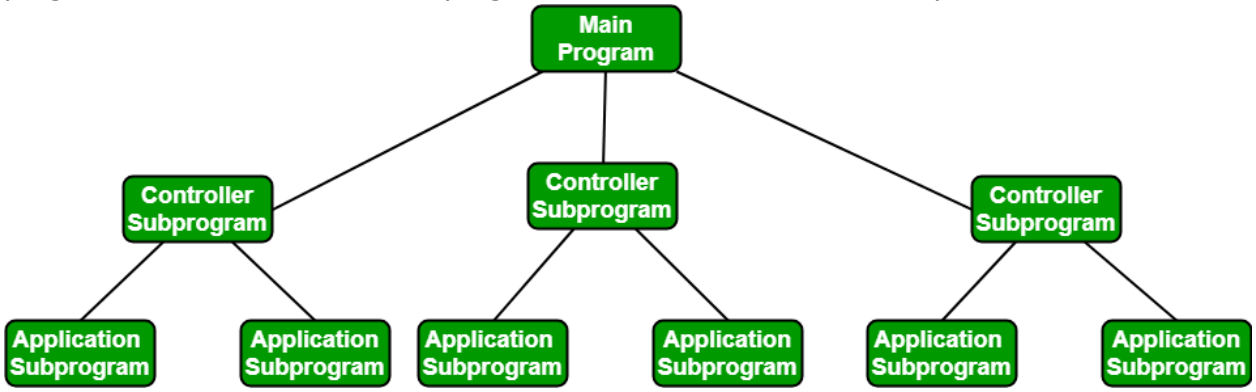


Fig: Software architecture styles

Object Oriented architecture: The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

Layered architecture:

A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.

At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)

Intermediate layers to utility services and application software functions.

Architectural Design

An architectural design performs the following functions.

1. It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.
2. It acts as a guideline for enhancing the system (when ever required) by describing those features of the system that can be modified easily without affecting the system integrity.
3. It evaluates all top-level designs.
4. It develops and documents top-level design for the external and internal interfaces.
5. It develops preliminary versions of user documentation.
6. It defines and documents preliminary test requirements and the schedule for software integration.

The sources of architectural design are listed below.

- regarding the application domain for the software to be developed
- Using data-flow diagrams
- Availability of architectural patterns and architectural styles.

Architectural Design Representation

Architectural design can be represented using the following models.

Structural model: Illustrates architecture as an ordered collection of program components
Dynamic model: Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment
Process model: Focuses on the design of the business or technical process, which must be implemented in the system
Functional model: Represents the functional hierarchy of a system
Framework model: Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.
Architectural Design Output

The architectural design process results in an Architectural Design Document (ADD). This document consists of a number of graphical representations thatcomprises software models along with associated descriptive text. The softwaremodels include static model, interface model, relationship model, and dynamic processmodel. They show how the system is organized into a process at run-time.

Architectural design document gives the developers a solution to the problem stated in the Software Requirements Specification (SRS). Note that it considers

only those requirements in detail that affect the program structure. In addition to ADD, other outputs of the architectural design are listed below.

Various reports including audit report, progress report, and configuration status accounts report
Various plans for detailed design phase, which include the following
Software verification and validation plan
Software configuration management plan
Software quality assurance plan
Software project management plan.

process and project Meyrics
metrics in the process and project domains

A software metric is a measure of software characteristics which are measurable or countable. Software metrics are valuable for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.

Within the software development process, many metrics are that are all connected. Software metrics are similar to the four functions of management: Planning, Organization, Control, or Improvement.

Classification of Software Metrics

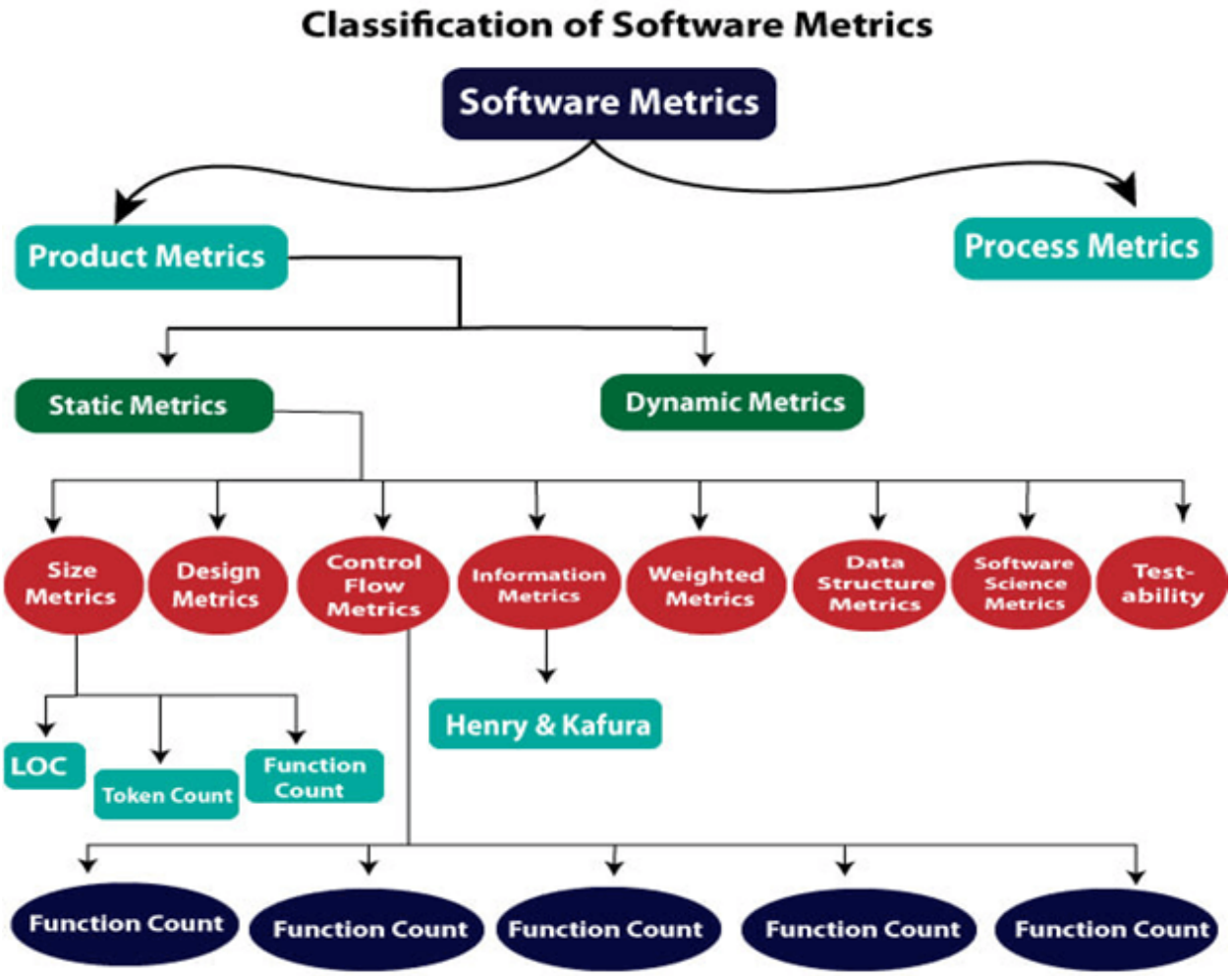
Software metrics can be classified into two types as follows:

1. Product Metrics: These are the measures of various characteristics of the software product. The two important software characteristics are:

1. Size and complexity of software.
2. Quality and reliability of software.

These metrics can be computed for different stages of SDLC.

2. Process Metrics: These are the measures of various characteristics of the software development process. For example, the efficiency of fault detection. They are used to measure the characteristics of methods, techniques, and tools that are used for developing software.



Types of Metrics

Internal metrics: Internal metrics are the metrics used for measuring properties that are viewed to be of greater importance to a software developer. For example, Lines of Code (LOC) measure.

External metrics: External metrics are the metrics used for measuring properties that are viewed to be of greater importance to the user, e.g., portability, reliability, functionality, usability, etc.

Hybrid metrics: Hybrid metrics are the metrics that combine product, process, and resource metrics. For example, cost per FP where FP stands for Function Point Metric.

Project metrics: Project metrics are the metrics used by the project manager to check the project's progress. Data from the past projects are used to collect various metrics, like time and cost; these estimates are used as a base of new software. Note that as the project proceeds, the project manager will check its progress from time-to-time and will compare the effort, cost, and time with the original effort, cost and time. Also understand that these metrics are used to decrease the development

costs, time efforts and risks. The project quality can also be improved. As quality improves, the number of errors and time, as well as cost required, is also reduced.

Advantage of Software Metrics

Comparative study of various design methodology of software systems.

For analysis, comparison, and critical study of different programming language concerning their characteristics.

In comparing and evaluating the capabilities and productivity of people involved in software development.

In the preparation of software quality specifications.

In the verification of compliance of software systems requirements and specifications.

In making inference about the effort to be put in the design and development of the software systems.

In getting an idea about the complexity of the code.

n taking decisions regarding further division of a complex module is to be done or not.

In guiding resource manager for their proper utilization.

In comparison and making design tradeoffs between software development and maintenance cost.

software measurement

A measurement is a manifestation of the size, quantity, amount, or dimension of a particular attribute of a product or process. Software measurement is a titrate impute of a characteristic of a software product or the software process. It is an authority within software engineering. The software measurement process is defined and governed by ISO Standard.

Software Measurement Principles:
The software measurement process can be characterized by five activities-

Formulation: The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
Collection: The mechanism used to accumulate data required to derive the formulated metrics.
Analysis: The computation of metrics and the application of mathematical tools.

Interpretation: The evaluation of metrics resulting in insight into the quality of the representation.
Feedback: Recommendation derived from the interpretation of product metrics transmitted to the software team.
Need for Software Measurement:
Software is measured to:

Create the quality of the current product or process.
Anticipate future qualities of the product or process.
Enhance the quality of a product or process.
Regulate the state of the project in relation to budget and schedule.
Classification of Software Measurement:
There are 2 types of software measurement:

Direct Measurement: In direct measurement, the product, process, or thing is measured directly using a standard scale.
Indirect Measurement: In indirect measurement, the quantity or quality to be measured is measured using related parameters i.e. by use of reference.
Metrics:
A metric is a measurement of the level at which any impute belongs to a system product or process.

Software metrics will be useful only if they are characterized effectively and validated so that their worth is proven. There are 4 functions related to software metrics:
Planning
Organizing
Controlling
Improving
Characteristics of software Metrics:
Quantitative: Metrics must possess quantitative nature. It means metrics can be expressed in values.
Understandable: Metric computation should be easily understood, and the method of computing metrics should be clearly defined.
Applicability: Metrics should be applicable in the initial phases of the development of the software.
Repeatable: The metric values should be the same when measured repeatedly and consistent in nature.
Economical: The computation of metrics should be economical.
Language Independent: Metrics should not depend on any programming language.
Classification of Software Metrics:
There are 3 types of software metrics:

Product Metrics: Product metrics are used to evaluate the state of the product, tracing risks and undercover prospective problem areas. The ability of the team to control quality is evaluated.
Process Metrics: Process metrics pay particular attention to enhancing the long-term process of the team or organization.
Project Metrics: The project matrix describes the project characteristic and execution process.

- Number of software developer
- Staffing patterns over the life cycle of software
- Cost and schedule
- Productivity

Metrics for software quality

Software metrics can be classified into three categories –

Product metrics – Describes the characteristics of the product such as size, complexity, design features, performance, and quality level.
Process metrics – These characteristics can be used to improve the development and maintenance activities of the software.

Project metrics – This metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics.

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.

Software quality metrics can be further divided into three categories –

- Product quality metrics
- In-process quality metrics
- Maintenance quality metrics
- Product Quality Metrics

This metrics include the following –

- Mean Time to Failure
- Defect Density
- Customer Problems
- Customer Satisfaction

Mean Time to Failure
It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

Defect Density
It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

Customer Problems

It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.

The problems metric is usually expressed in terms of Problems per User-Month (PUM).

PUM = Total Problems that customers reported (true defect and non-defect oriented problems) for a time period + Total number of license months of the software during the period
Where,

Number of license-month of the software = Number of install license of the software ×
Number of months in the calculation period
PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year.

Customer Satisfaction
Customer satisfaction is often measured by customer survey data through the five-point scale −

Very satisfied
Satisfied
Neutral
Dissatisfied
Very dissatisfied
Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example −

Percent of completely satisfied customers
Percent of satisfied customers
Percent of dis-satisfied customers
Percent of non-satisfied customers
Usually, this percent satisfaction is used.

In-process Quality Metrics
In-process quality metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes −

Defect density during machine testing
Defect arrival pattern during machine testing
Phase-based defect removal pattern
Defect removal effectiveness
Defect density during machine testing
Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. Higher defect rates

found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort.

This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect arrival pattern during machine testing
The overall defect density during testing will provide only the summary of the defects. The pattern of defect arrivals gives more information about different quality levels in the field. It includes the following –

The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.

The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.

The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

Phase-based defect removal pattern
This is an extension of the defect density metric during testing. In addition to testing, it tracks the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.

Because a large percentage of programming defects is related to design problems, conducting formal reviews, or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management.

Defect removal effectiveness
It can be defined as follows –

DRE=DefectremoveddduringadevelopmentphaseDefectslatentintheproduct×100%
This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called early defect removal when used for the front-end and phase effectiveness for specific phases. The higher the

value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.

Maintenance Quality Metrics
Although much cannot be done to alter the quality of the product during this phase, following are the fixes that can be carried out to eliminate the defects as soon as possible with excellent fix quality.

Fix backlog and backlog management index
Fix response time and fix responsiveness
Percent delinquent fixes
Fix quality
Fix backlog and backlog management index
Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.

Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.

BMI=NumberofproblemsclosedduringthemonthNumberofproblemsarrivedduringthemonth×100%
If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

Fix response time and fix responsiveness
The fix response time metric is usually calculated as the mean time of all problems from open to close. Short fix response time leads to customer satisfaction.

The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer.
Percent delinquent fixes
It is calculated as follows –

Percent Delinquent Fixes = Number of fixes that exceeded the response time criteria by ceverity level Number of fixes delivered in a specified time×100%Fix Quality
Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.

A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure; the second is a process measure. The difference between the two dates is the latent period of the defective fix.

Usually the longer the latency, the more will be the customers that get affected. If the number of defects is large, then the small value of the percentage metric will show an optimistic picture. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.

MOHAN BABU UNIVERSITY
Sree Sainath Nagar, Tirupati 517 102

MODULE-4 SOFTWARE TESTING STRATEGIES AND APPLICATIONS

Testing strategies: A strategic approach to software testing, trategic issues, Test strategies for conventional software, Test strategies for object oriented software, Validation testing, System testing, The art of debugging.
Testing Conventional Applications: Software testing fundamentals, White box testing-Basis path testing, Control structure testing; Black box testing, Object oriented testing methods

TESTING STRATEGIES

A Strategic Approach to Software Testing:

* Testing is a group of tasks that can be methodically and premeditatedly completed.

* A collection of testing procedures and test case design strategies are explained for a software process.

* Several testing techniques are suggested

* Diverse testing techniques are suggested

* Testing is a series of tasks that can be organized and completed methodically in advance.

Testing Strategies – Generic Characteristics:

* A group of software developers must conduct efficient formal technical reviews before they may test software.

* Testing starts with the individual components and progresses to the integration of the complete computer-based system.

* The software developer and, for larger projects, an independent test group carry out the testing.

* Testing begins with component level work and progresses to the integration of the entire computer-based system.

****

Note:

Testing VS Debugging:

While testing and debugging are two different processes, every testing approach must include debugging.

* A testing strategy should include guidelines for the tester as well as a management checklist of benchmarks.

****

(1) Verification and Validation:

* Verification refers to a series of steps taken to ensure that a programme correctly accomplishes a specified task.

Example:

Verification: Are we building the product right?

* Various procedures are carried out under the umbrella term of "validation" to guarantee that the final product of software development is tied to the requirements of the customer.

Example:

Validation: Are we building the right product?

* The processes of verification and validation involve a wide range of SQA operations that include the following:

=> Formal Technical Reviews
=> Quality and Configuration audits
=> Performance Monitoring
=> Simulation
=> Feasibility Study
=> Documentation Review
=> Database Review
=> Analysis Algorithm
=> Development Testing
=> Usability Testing
=> Qualification Testing
=> Installation Testing

(2) Organizing for Software Testing:

* The developer often also carries out integration testing, which is a phase of testing that comes before the complete software architecture is constructed.

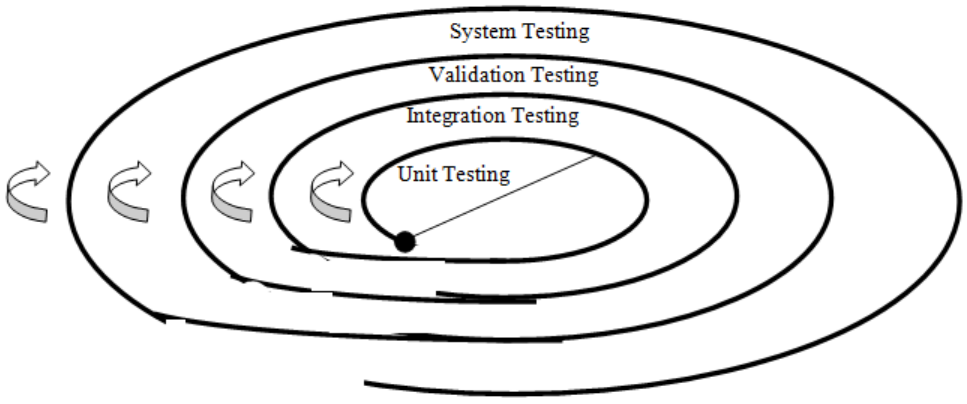
* Testing the various units (components) of the program is always the responsibility of the software developer.

* Once the software architecture has been completed, an independent testing group will be recruited to evaluate the product. The purpose of an Independent Test Group, which is also abbreviated as an ITG, is to eliminate the inherent challenges that come when the builder is given the opportunity to test the thing that has been built. This is accomplished by eliminating the inherent difficulties. During the course of a software project, the developer and the ITG work together very closely to ensure that through testing will be performed.

* The developer needs to be available when testing is being done so that he or she may fix any mistakes that are found.

(3) Software testing strategy for conventional software architecture:

* A testing method for software can be understood in the context of the spiral, as depicted in the following diagram:



Testing Strategy

* Unit testing starts at the middle of the spiral and concentrates on the software's components as they are written in the source code. * Proceeding in a different direction down the spiral, we encounter integration testing, which is concerned with the planning and building of software architecture. * Next, validation testing is introduced, which compares requirements that have been established during software requirements analysis

with software that has already been developed. * Finally, we come across verification testing, which verifies that software that has been constructed satisfies requirements that were determined during software requirements At last, we come to the system testing phase, which involves testing the software along with the other components of the system as a whole.

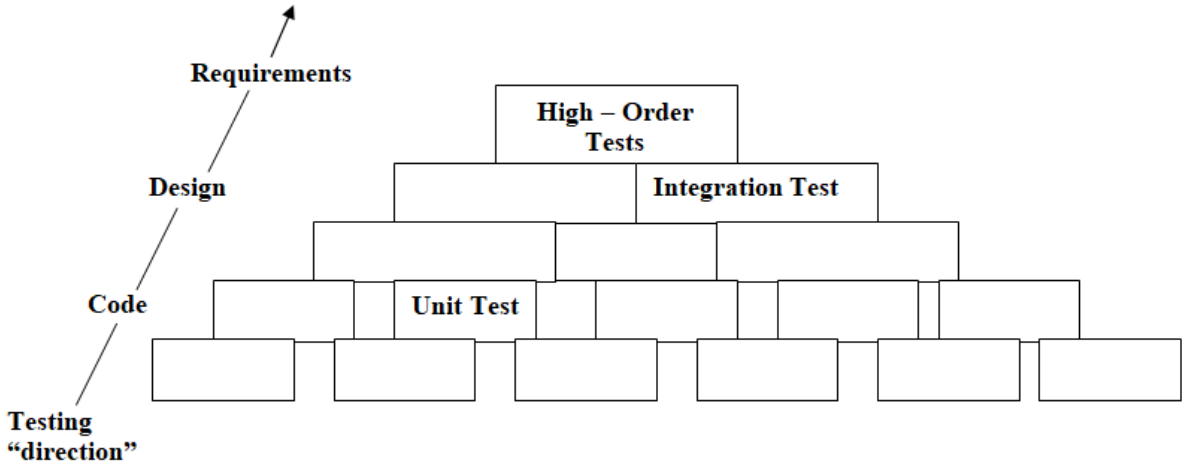
Software Testing Steps:

(i) Unit Testing:

* The initial phase of testing involves the examination of each component in isolation to verify its proper functioning as an independent unit.

* Unit testing employs rigorous testing techniques to achieve comprehensive coverage and optimal error detection within the control structure of the components.

* Subsequently, the components are integrated to form cohesive software packages.



(ii) Integration Testing:

* It solves the problems that arise when verifying and building a programme simultaneously

* The requirements analysis validation criteria

* need to be tested. With validation testing, you can rest assured that your software will perform as expected in every way.

* A series of high - order tests are executed after the software has been integrated [built].

(iii) High Order Testing:

* It is not considered to be a part of software engineering

* After the programme has been verified, it needs to be integrated with the other components of the system (for example, the hardware, the people, and the software).

* Testing a system ensures that all of its components work together as intended and that the desired level of overall functionality and performance is reached.

6.4 Strategic Issues:

* If you want your software testing strategy to be successful, you need to address the following issues:

(1) Specify product requirements in a quantifiable manner well in advance of testing beginning;

(2) State testing objectives explicitly;

(3) Understand who will be using the software and create a profile for each user category.

(4) Formulate a strategy for tasting that places an emphasis on "Rapid Cycle Testing."

(5) Construct reliable software that is intended to perform its own testing.

(6) Employ efficient formal technical reviews as a filter prior to testing.

(7) Carry out formal technical reviews to evaluate the test strategy and test cases.

(8) Construct an approach for the testing process that emphasises continual improvement.

Test Strategy for Conventional Software:

* There are many different ways to test software, thus some of the available options are as follows:

(i) A software team could wait until the system is completely built, and then they could run tests on the system to look for faults. * This strategy is not effective in the majority of situations.

(ii) A software engineer might be able to run tests every day anytime any component of the system is being built

* This strategy has the potential to yield excellent results. But the majority of software engineers are reluctant to make use of it

(iii) The majority of software teams opt for a testing technique that lies somewhere in the middle of the two extremes

It adopts an incremental approach to testing, beginning with the testing of individual programme units, then moving on to tests meant to aid the integration of the modules, and finally culminating with tests that exercise the created system as a whole.

6.6 Unit Testing:

* It concentrates on the verification of the smallest unit of software design (also known as a software component or module);

* It concentrates on the internal processing logic and data structures that are contained inside the bounds of the component;

Unit test Considerations:

* The test that is performed as a part of the unit tests is displayed below

Interface:

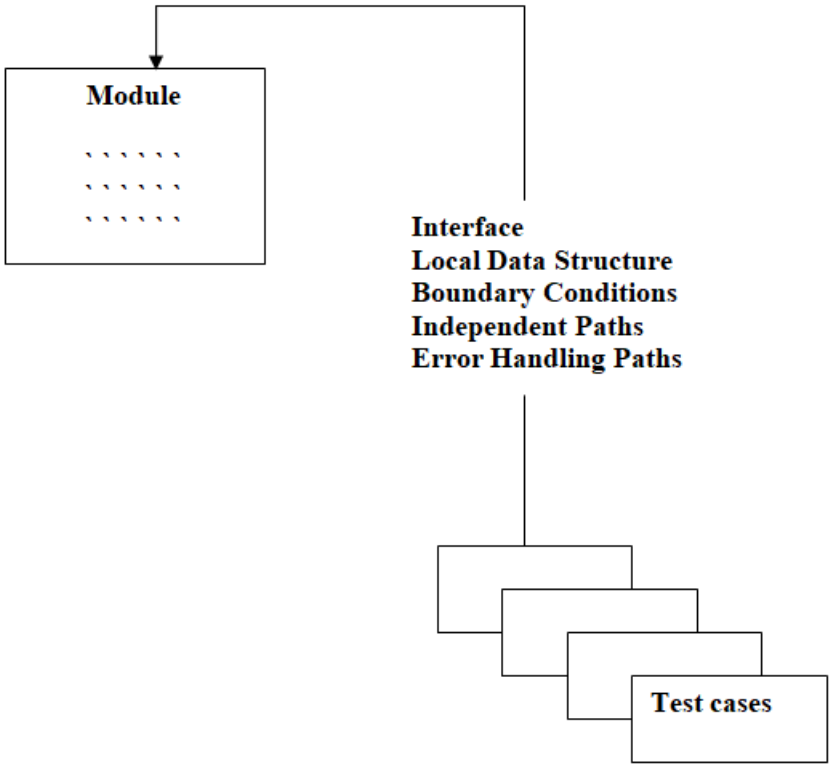
* It is examined to ensure the following conditions are met:

=> Information is able to flow correctly into and out of the programme unit that is being evaluated Local

Data structures:

* These are investigated to guarantee that

=> The data that is kept in a temporary storage space will keep its integrity during the entirety of the operation of an algorithm.



Independent Paths:

* Each and every basis path that passes through the control structures is investigated to guarantee that

=> Each and every statement contained within a module has been run at least once.

Boundary Conditions:

* These are verified to ensure that

=> the module runs correctly within the boundaries that have been specified in order to limit (Or restrict) processing.

* And lastly, all possible error-handling routes are put through their paces.

Before any other tests can be run, there must first be validation of the dataflow across an interface module.

another test will be performed.

During the unit testing process, one of the most important tasks is to perform selective testing of the execution path.

* Test cases must to be developed to unearth faults brought on by= Incorrect Mathematical Processing Comparisons That Are Not Accurate Control not being properly maintained

Common errors in computations are:

(1) Mixed mode operations

(2) Misunderstood arithmetic precedence

(3) wrong initialization

(4) Precision inaccuracy

(5) Misunderstood or wrong arithmetic precedence

(6) An inaccurate portrayal of a symbol representing an expression

Test cases should uncover errors, such as

(1) A comparative analysis of multiple data formats;

(2) erroneous logical operators (Or) hierarchy

(3) Expecting equality when precision error makes it unlikely;

(4) Making incorrect variable comparisons

(5) Incorrect (Or nonexistent) loop termination;

(6) Failure to terminate when divergent iteration occurs;

(7) Inaccurately changed loop variables

Boundary Testing:

* This is one of the most significant responsibilities involved in unit testing

* A common cause of software failure is when it reaches one of its limits (for example, an error frequently happens when the nth element of an n-dimensional array is handled). When evaluating error handling, the following are some examples of potential errors that should be tested:

Processing under exception conditions is not right. The error description is insufficient to help identify the error's cause.

(1) The error description is not clear.

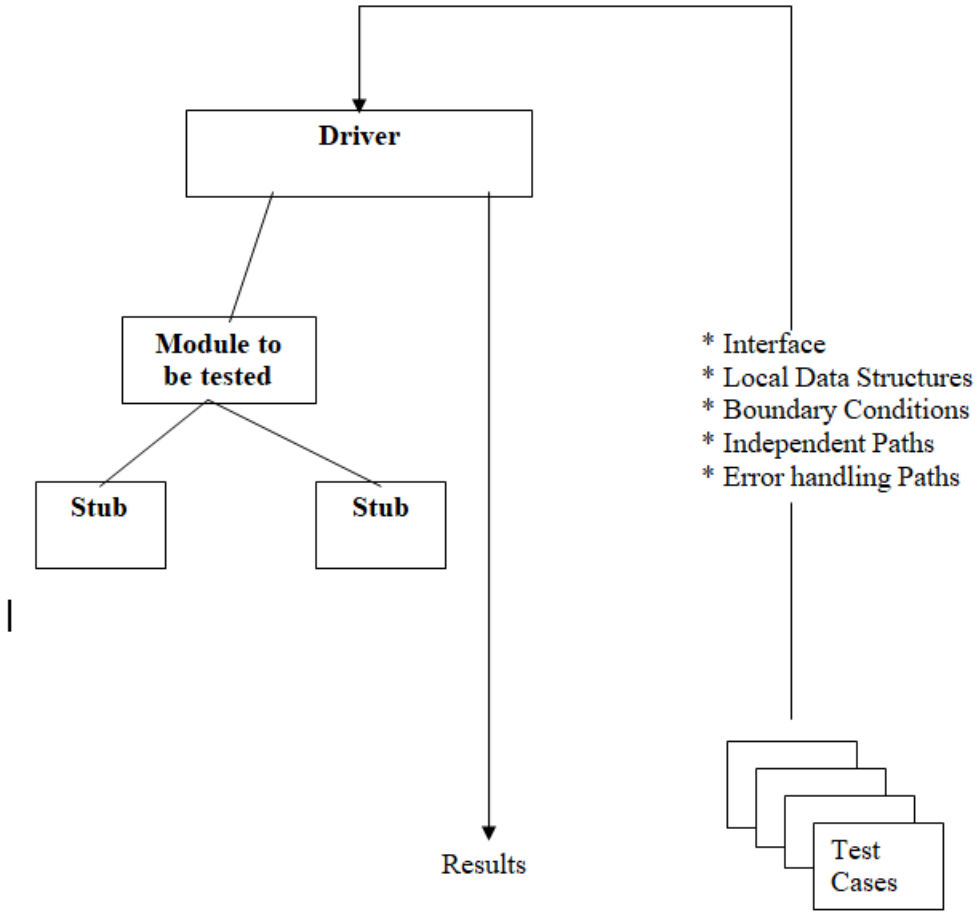
(2) The error reported does not match the error experienced.

(3) An operating system intervention occurs before error handling due to an error state.

(4) The error description is insufficient to help identify the error's cause.

Unit Test Procedures:

* Unit test design can be done either before or after code is generated.



Driver:

* A driver is not much more than an application's "main programme" in the vast majority of cases * It accepts

=> data from test cases

=> Sends these data to the component that is [about to be tested].

=> Print the findings that are relevant.

Stub (Or) Dummy Programs:

* It takes the place of modules that are called by the component that is being tested.
* Stub utilises
=> the interface of the subordinate module.
= >Manipulate the data as little as possible
.=> guarantees the authenticity of the entry
=> transfers control back to the module that is currently being evaluated

* Drivers and stubs are two different types of software that need to be built, but they are not included in the final software product.
* The real overhead is reasonably modest if the drivers and stubs are kept simple; otherwise, it is substantial.
* When a component that has a high cohesiveness is designed, it simplifies the unit testing process.

Integration Testing:

* Once all of the modules have passed their own unit tests meaning that all of the modules function properly, we have doubts about whether or not they will work, when do we integrate them together?
* Integration testing is going to be the solution for this problem.

Interfacing:

It is the mechanism that brings together all of the individual modules.
The following are the issues that can arise throughout the interfacing process: It is possible to lose data when moving between interfaces.
=> An unintended negative effect can be caused by one module on another module.
=> The combination of subfunctions might not result in the principal function that is required.

What it does

=> An imprecision that is tolerable on its own might become intolerable when it is multiplied.

scales levels

=> Issues may arise due to the use of global data structures

Integration Testing – Definition:

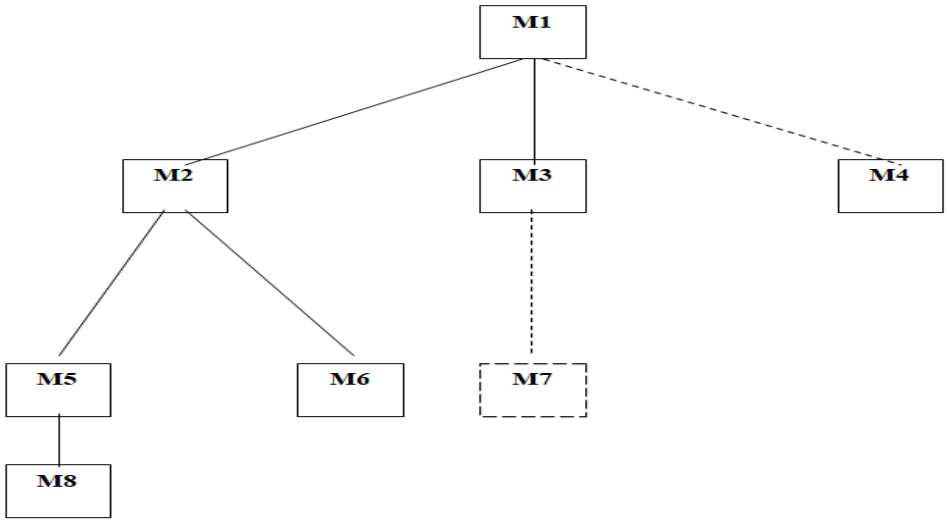
* It is a methodical approach to building the software architecture, while at the same time running tests to find faults linked with the software's interface.
* The goal is to use components that have undergone unit testing and construct a programme structure according to the specifications set out by the design.

Incremental Integration:

* In this scenario, the programme is built and tested in small steps
* Errors are easy to localise and rectify
* Interfaces are tested in their entirety and a systematic testing strategy may be used
* There are several variants of the incremental integration method to choose from.

Incremental Integration – Types:

(1) Top – Down Integration:



The software architecture is being built up in stages, including through this testing, which is an incremental method.

* The modules are integrated by starting with the primary control module (also known as the main programme) and progressing downwards down the control hierarchy.
* The subordinate module to the primary control module can be incorporated in either a breadth first or a depth first fashion.

Depth First Integration:

* The program structure's primary control path is integrated by all of its components.
* The primary route chosen is determined by the features unique to the application.

* For instance, if the left-hand path components were chosen, integration would begin with M1, M2, and M5, and finish with M8.
* Subsequently, the right hand and center control routes are built.

Breadth First integration:

* It moves the structure horizontally by incorporating all elements that are directly subordinate at each level.

The subassemblies M2, M3, and M4 in the picture on the right would be integrated first, then M5, M6, and so on.

The following are the steps that make up the integration process:

Step 1: the main control module serves as the test driver, and stubs are installed in place of all components that are immediately subordinate to the main control module.

Step 2: Subordinate stubs are replaced one at a time with genuine components, and this process is carried out in one of two ways, determined by the integration strategy that was chosen: breadth first or depth first.

Step3 :, tests are carried out while each component is being merged.

Step 4: Once each group of duties has been finished, a new stub will be removed and replaced with the actual component.

Step 5: You may want to perform regression testing in order to make certain that no new mistakes have been introduced.

* Beginning with Step 2, the procedure will continue until the entirety of the programme structure is developed.

Advantages:

(1) Top-down integration ensures that significant control or decision points are validated at an earlier stage in the testing process, which is beneficial in a number of ways.

(2) It is to the advantage of both the customer and the developer to conduct an early demonstration of the product's functional capabilities in order to build the customer's confidence and the developer's. This is significant since it reveals that the feature is performing as planned, which is important information to have.