UNIT-4

TESTING STRATEGIES

Strategic Approach to software testing:

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test case design techniques and testing methods—should be defined for the software process.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

• Testing begins at the component level2 and works "outward" toward the integration of the entire computer-based system.

• Different testing techniques are appropriate at different points in time.

• Testing is conducted by the developer of the software and (for large projects) an independent test group.

• Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

- **Verification and Validation**

  Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

  Verification:"Are we building the product right?"
  Validation: "Are we building the right product?"

- **Organizing for Software Testing**

  For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigate against thorough testing.

There are often a number of misconceptions that can be erroneously inferred from the preceeding discussion: **(1)** that the developer of software should do no testing at all, **(2)** that the software should be "tossed over the wall" to strangers who will test it mercilessly, **(3)** that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

Test strategies for Conventional Software

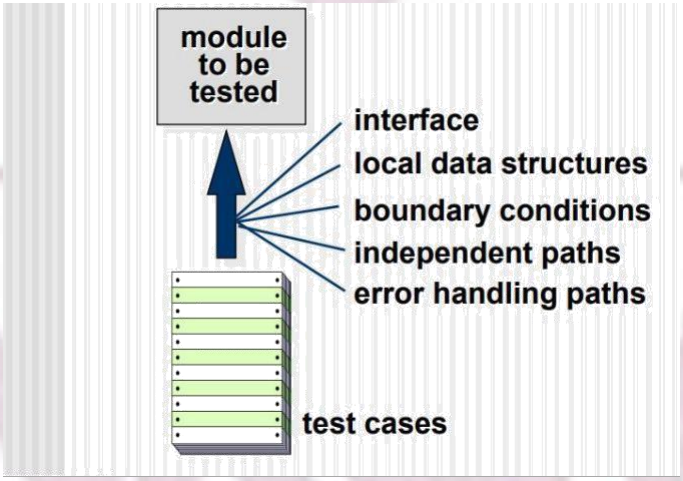
- There are many strategies that can be used to test software.
- At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
- This approach simply does not work. It will result in buggy software.
- At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.
- This approach, although less appealing to many, can be very effective.
- A testing strategy that is chosen by most software teams falls between the two extremes.
- It takes an incremental view of testing,
- Beginning with the testing of individual program units,
- Moving to tests designed to facilitate the integration of the units,
- Culminating with tests that exercise the constructed system.

Unit Test :

Unit testing focuses verification effort on the smallest unit of software design—the software component or module.

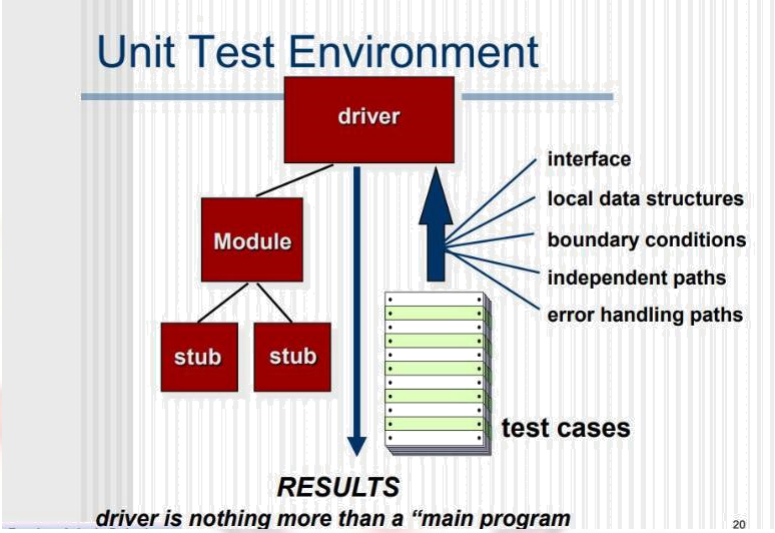
The unit test focuses on the internal processing logic and data structures within the boundaries of a component.

This type of testing can be conducted in parallel for multiple components.



- Unit tests are illustrated schematically in previous Figure.
- **The module interface** is tested to ensure that information properly flows into and out of the program unit under test.
- **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.

- **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- **Finally, all error-handling paths are tested**
- Good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur.



Integration Testing:

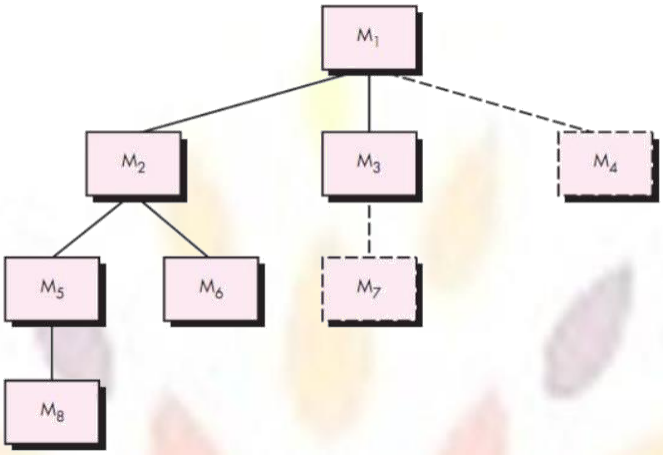
Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

Different Integration Testing Strategies :

- Top-down testing
- Bottom-up testing
- Regression Testing
- Smoke Testing

Top-down testing

- Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to **the main control module are incorporated into the structure in either a *depth-first or breadth-first manner.***



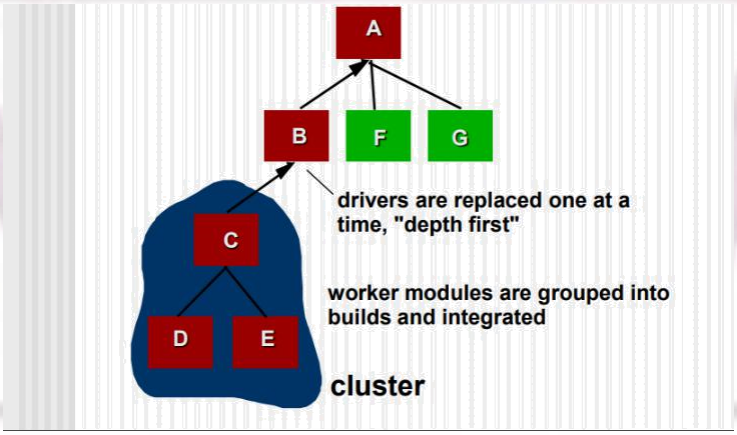
BOTTOM-UP INTEGRATION TESTING:

Bottom-up integration testing, It begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps…

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.



REGRESSION TESTING:

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

It is impractical and inefficient to reexecute every test for every program function once a change has occurred….

Regression testing is a type of software testing that seeks to uncover new software bugs, OR

Regression testing is the process of testing, changes to computer programs to make sure that the older programming still works with the new changes. Here changes such as enhancements, patches or configuration changes, have been made to them.

SMOKE TESTING:

Smoke testing is an integration testing approach that is commonly used when product software is developed

Smoke testing is performed by developers before releasing the build to the testing team and after releasing the build to the testing team it is performed by testers whether to accept the build for further testing or not.

It is designed as a pacing (Speedy) mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

Smoke testing provides a number of benefits when it is applied on complex, time critical software projects.

- I**ntegration risk is minimized.** Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early.
- **The quality of the end product is improved**. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- **Error diagnosis and correction are simplified.**
- **Progress is easier to assess**

Black – box and white – box testing

Black – box testing:

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

➢ **Types of Black Box Testing**

There are many types of Black Box Testing but the following are the prominent ones –

- **Functional testing** – This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** – This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** – Regression Testing **is** done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

White – box testing:

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**.

It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing.

In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software.

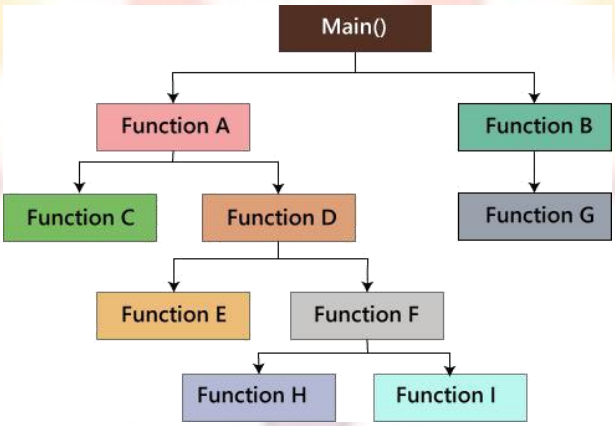
The white box testing contains various tests, which are as follows:

- Path testing

- Loop testing
- Condition testing
- Testing based on the memory perspective
- Test performance of the program

Path testing---

In the path testing, we will write the flow graphs and test all independent paths. Here writing the flow graph implies that flow graphs are representing the flow of the program and also show how every program is added with one another as we can see in the below image:



Loop testing---

In the loop testing, we will test the loops such as while, for, and do-while, etc. and also check for ending condition if working correctly and if the size of the conditions is enough.

Condition testing---

In this, we will test all logical conditions for both **true** and **false** values; that is, we will verify for both **if** and **else** condition.

Testing based on the memory (size) perspective---

The size of the code is increasing for the following reasons:

- **The reuse of code is not there**: let us take one example, where we have four programs of the same application, and the first ten lines of the program are similar. We can write these ten lines as a discrete function, and it should be accessible by the above four programs as well. And also, if any bug is there, we can modify the line of code in the function rather than the entire code.

- The **developers use the logic** that might be modified. If one programmer writes code and the file size is up to 250kb, then another programmer could write a similar code using the different logic, and the file size is up to 100kb.
- The **developer declares so many functions and variables** that might never be used in any portion of the code. Therefore, the size of the program will increase.

Test the performance (Speed, response time) of the program---

The application could be slow for the following reasons:

- When logic is used.
- For the conditional cases, we will use **or** & **and** adequately.
- Switch case, which means we cannot use **nested if**, instead of using a switch case.

Differences between Black Box Testing vs White Box Testing:

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
Implementation of code is not needed for black box testing.	Code implementation is necessary for white box testing.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as outer or external software testing.	It is the inner or the internal software testing.
It is functional test of the software.	It is structural test of the software.
This testing can be initiated on the basis of requirement specifications document.	This type of testing of software is started after detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the	It is the logic testing of the software.

Black Box Testing	White Box Testing
software.	
➢ It is applicable to the higher levels of testing of software.	➢ It is generally applicable to the lower levels of software testing.
➢ It is also called closed testing.	➢ It is also called as clear box testing.
➢ It is least time consuming.	➢ It is most time consuming.
➢ It is not suitable or preferred for algorithm testing.	➢ It is suitable for algorithm testing.
➢ Can be done by trial and error ways and methods.	➢ Data domains along with inner or internal boundaries can be better tested.
➢ **Example:** search something on google by using keywords	➢ **Example:** by input to check and verify loops
➢ **Types of Black Box Testing:** ➢ A. Functional Testing ➢ B. Non-functional testing ➢ C. Regression Testing	➢ **Types of White Box Testing:** ➢ A. Path Testing ➢ B. Loop Testing ➢ C. Condition testing

Difference between Alpha and Beta Testing:

Alpha Testing	Beta Testing
➢ Alpha testing involves both the white box and black box testing.	➢ Beta testing commonly uses black-box testing.
➢ Alpha testing is performed by testers who are usually internal employees of the organization.	➢ Beta testing is performed by clients who are not part of the organization.
➢ Alpha testing is performed at the developer's site.	➢ Beta testing is performed at the end-user of the product.
➢ Reliability and security testing	➢ Reliability, security and robustness are

Alpha Testing	Beta Testing
are not checked in alpha testing.	checked during beta testing.
Alpha testing ensures the quality of the product before forwarding to beta testing.	Beta testing also concentrates on the quality of the product but collects users input on the product and ensures that the product is ready for real time users.
Alpha testing requires a testing environment or a lab.	Beta testing doesn't require a testing environment or lab.
Alpha testing may require a long execution cycle.	Beta testing requires only a few weeks of execution.
Developers can immediately address the critical issues or fixes in alpha testing.	Most of the issues or feedback collected from the beta testing will be implemented in future versions of the product.
Multiple test cycles are organized in alpha testing.	Only one or two test cycles are there in beta testing.

Validation Testing

The definition of validation testing in software engineering is in place to determine if the existing system complies with the system requirements and performs the dedicated functions for which it is designed along with meeting the goals and needs of the organization.

This mode of testing is extremely important especially if you want to be ***one of the best software testers***. ***The*** software verification and validation testing is the process after the validation testing stage is secondary to verification testing.

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*

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.

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.

System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system).

The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware.

System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.

Types of system test:

- Recovery testing
- Security testing
- Stress testing
- Performance testing

Recovery testing:

It is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

If recovery is automatic, reinitialization, check pointing, mechanisms, data recovery, and restart and evaluated for correctness.

Security testing:

Verifies the protection mechanisms built into a system will.

Stress testing:

It executes a system in a manner that demands resources in abnormal quality, frequency, or volume.

Performance testing:

It designed to test the run – time performance of software with in the context of an integrated system.

The art of debugging

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.

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.

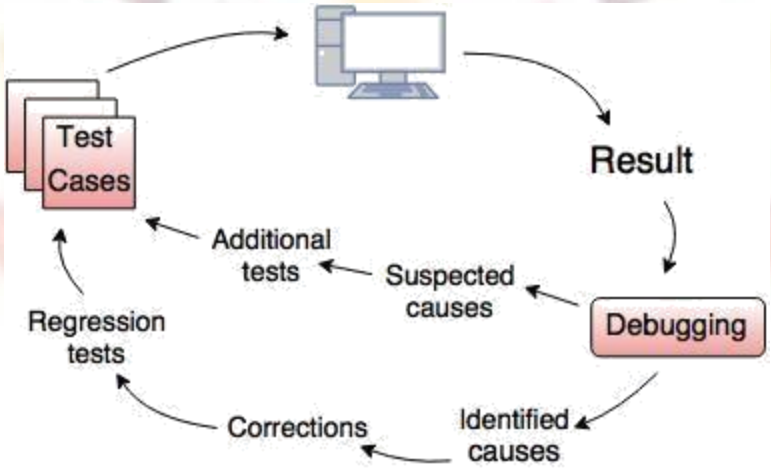


Fig. - Debugging process

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.

4.2. Product metrics

Software quality

Software quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

McCall's quality Factors

According to McCall's model, product operation category includes five software quality factors, which deal with the requirements that directly affect the daily operation of the software. They are as follows –

Correctness:

These requirements deal with the correctness of the output of the software system. They include –

- Output mission
- The required accuracy of output that can be negatively affected by inaccurate data or inaccurate calculations.
- The completeness of the output information, which can be affected by incomplete data.

Reliability:

Reliability requirements deal with service failure. They determine the maximum allowed failure rate of the software system, and can refer to the entire system or to one or more of its separate functions.

Efficiency: It deals with the hardware resources needed to perform the different functions of the software system.

Integrity: This factor deals with the software system security, that is, to prevent access to unauthorized persons, also to distinguish between the group of people to be given read as well as write permit.

Usability: Usability requirements deal with the staff resources needed to train a new employee and to operate the software system.

Maintainability: This factor considers the efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of the corrections.

Flexibility: This factor deals with the capabilities and efforts required to support adaptive maintenance activities of the software.

Testability: Testability requirements deal with the testing of the software system as well as with its operation.

Portability: Portability requirements tend to the adaptation of a software system to other environments consisting of different hardware, different operating systems, and so forth.

Reusability: This factor deals with the use of software modules originally designed for one project in a new software project currently being developed.

Interoperability: Interoperability requirements focus on creating interfaces with other software systems or with other equipment firmware.

ISO 9126 QUALITY FACTORS:

1. **Functionality:** The functions are those that will satisfy implied needs.
   - Suitability
   - Accuracy
   - Interoperability
   - Security
   - Functionality Compliance
2. **Reliability:** A set of attributes that will bear on the capability of software to maintain the level of performance.
   - Maturity
   - Fault Tolerance
   - Recoverability
   - Reliability Compliance
3. **Usability:** A set of attributes that bear on the effort needed for use by a implied set of users.
   - Understandability
   - Learn ability
   - Operability
   - Attractiveness
   - Usability Compliance
4. **Efficiency:** A set of attributes that bear on the relationship between the level of performance of the software under stated conditions.
   - Time Behavior
   - Resource Utilization
   - Efficiency Compliance
5. **Maintainability:** A set of attributes that bear on the effort needed to make specified modifications.
   - Analyzability
   - Changeability
   - Stability
   - Testability
   - Maintainability Compliance
6. **Portability:** A set of attributes that bear on the ability of software to be transferred from one environment to another.
   - Adaptability
   - Installability

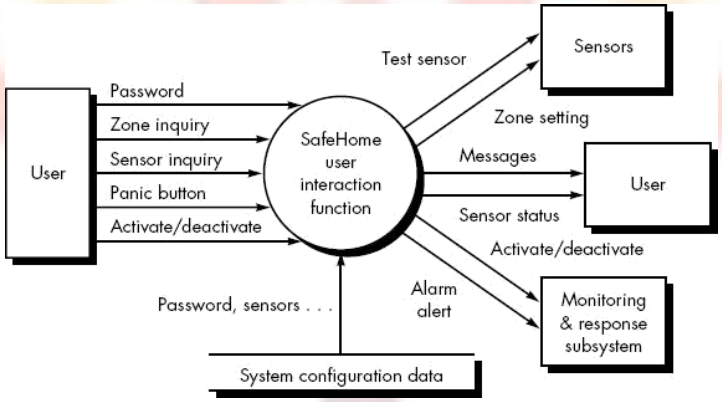
- Co-existence
- Replace ability
- Portability Compliance

Metrics for Analysis model

Technical work in software engineering begins with the creation of the analysis model. It is at this stage that requirements are derived and that a foundation for design is established. Therefore, technical metrics that provide insight into the quality of the analysis model are desirable.

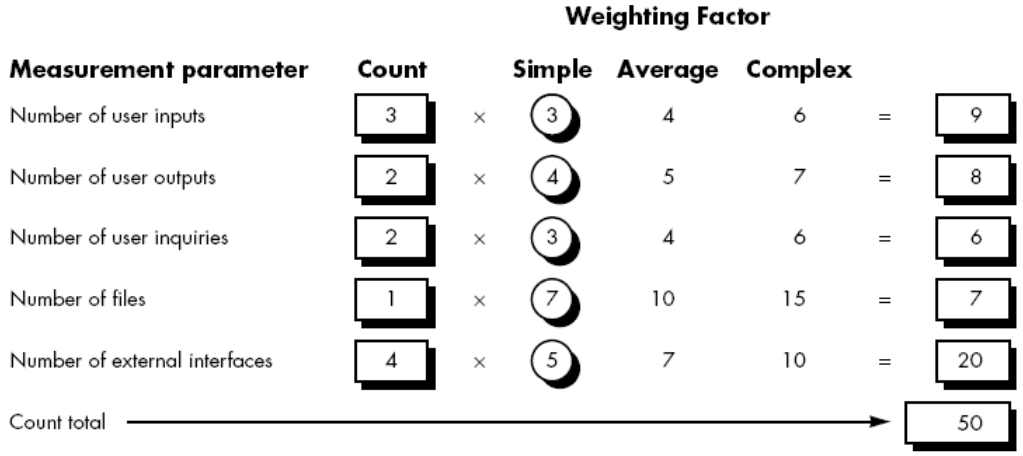
Function-Based Metrics:

The function point metric an be used effectively as a means for predicting the size of a system that will be derived from the analysis model.



The data flow diagram is evaluated to determine the key measures required for computation of the function point metric :

• number of user inputs
• number of user outputs
• number of user inquiries
• number of files
• number of external interfaces



The count total **FP = count total [0.65 + 0.01 (Fi)]**

where count total is the sum of all FP entries obtained from the first figure and Fi (i = 1 to 14) are "complexity adjustment values."

Metrics for Specification Quality

Davis and his colleagues propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: specificity (lack of ambiguity), completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability.

- we assume that there are nr requirements in a specification, such that **nr = nf + nnf**

where nf is the number of functional requirements and nnf is the number of nonfunctional (e.g., performance) requirements.

- To determine the specificity (lack of ambiguity) of requirements, Davis et al. suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

Q1 = nui/nr

where nui is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

- The completeness of functional requirements can be determined by computing the ratio

Q2 = nu/[ni x ns]

where nu is the number of unique function requirements, ni is the number of inputs (stimuli) defined or implied by the specification, and ns is the number of states specified. The Q2 ratio measures the percentage of necessary functions that have been specified for a system.

- **Q3 = nc/[nc + nnv]**

where nc is the number of requirements that have been validated as correct and nnv is the number of requirements that have not yet been validate

Metrics for design model

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative .

1. Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass define three software design complexity measures: structural complexity, data complexity, and system complexity.

Structural complexity of a module i is defined in the following manner:

S(i) = f 2 out(i)

where fout(i) is the fan-out7 of module i.

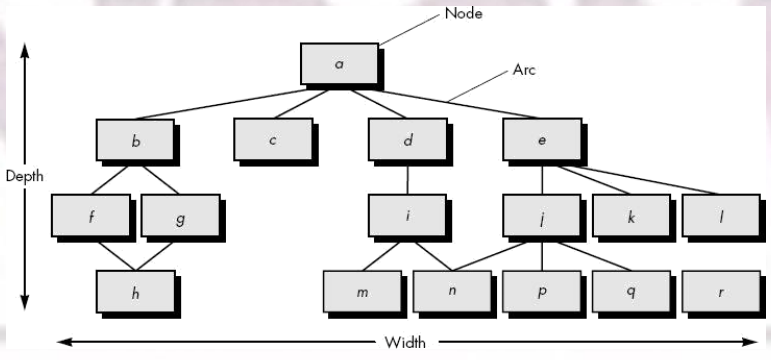
Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

D(i) = v(i)/[fout(i) +1]

where v(i) is the number of input and output variables that are passed to and from module i.

Finally, system complexity is defined as the sum of structural and data complexity, specified as

C(i) = S(i) + D(i)



size = n + a

where n is the number of nodes and a is the number of arcs. For the architecture shown in figure,

size = 17 + 18 = 35

depth = the longest path from the root (top) node to a leaf node. For the architecture shown infigure, depth = 4.
width = maximum number of nodes at any one level of the architecture. For the architecture shown in figure, width = 6.
arc-to-node ratio, r = a/n,

the Air Force uses information obtained from data and architectural design to derive a design structure quality index (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI :

S1 = the total number of modules defined in the program architecture.
S2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S2).
S3 = the number of modules whose correct function depends on prior processing.
S4 = the number of database items (includes data objects and all attributes that define objects).
S5 = the total number of unique database items.
S6 = the number of database segments (different records or individual objects).
S7 = the number of modules with a single entry and exit (exception processing is not considered to be a multiple exit).

Once values S1 through S7 are determined for a computer program, the following intermediate values can be computed:

Program structure: D1, where D1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then D1 = 1, otherwise D1 = 0.
Module independence: D2 = 1 (S2/S1)
Modules not dependent on prior processing: D3 = 1 (S3/S1)
Database size: D4 = 1 (S5/S4)
Database compartmentalization: D5 = 1 (S6/S4)
Module entrance/exit characteristic: D6 = 1 (S7/S1)
With these intermediate values determined, the DSQI is computed in the following manner:

DSQI = wiDi

where i = 1 to 6, wi is the relative weighting of the importance of each of the intermediate values, and wi = 1 (if all Di are weighted equally, then wi = 0.167).

2. Metrics for object – oriented design

- Size
- Complexity
- Coupling
- Sufficiency
- Completeness

- Cohesion
- Primitiveness
- Similarity
- Volatility

Metrics for source code

HSS(Halstead Software science)

Primitive measure that may be derived after the code is generated or estimated once design is

Complete.

n1 = the number of distinct operators that appear in a program

n2 = the number of distinct operands that appear in a program

N1 = the total number of operator occurrences.

N2 = the total number of operand occurrence.

Overall program length N can be computed:

N = n1 log2 n1 + n2 log2 n2

V = N log2(n1+ n2)

V will vary with programming language and represent the volume of information required to specify a program.

Halstead defines a volume ratio L as the ratio of volume of the most compact from of a program to the volume of the actual program. In actuality, L must be less than 1.

In terms of primitive measures, the volume ratio may be expressed

as L=2/n1 * n2/N2

METRIC FOR TESTING

- Halstead metrics applied to testing:

n1 = the number of distinct operators that appear in a program

n2 = the number of distinct operands that appear in a program

N1 = the total number of operator occurrences.

N2 = the total number of operand occurrence.

Program Level and Effort

PL = 1/[(n1 / 2) x (N2 / n2 l)]

e = V/PL

- Metrics for object oriented testing
- Lack of cohesion in method(LCOM)
- Percent public and protected(PAP)
- public access to data members(PAD)
- Number of root classes(NOR)
- Fan-in (FIN)

METRICS FOR MAINTENANCE

Mt = the number of modules in the current release

Fc = the number of modules in the current release that have been changed

Fa = the number of modules in the current release that have been added.

Fd = the number of modules from the preceding release that were deleted in the current release

The Software Maturity Index, SMI, is defined as:

SMI = [Mt–(Fc + Fa +Fd)/ Mt]