UNIT- 4

SOFTWARE TESTING TECHNIQUE:

Fundamental Principal of Testing:

The objective of the testing is to provide a quality product to the customer.

1. The goal of testing is to find defects before customers find them out.
2. Exhaustive testing is not possible; program testing can only show the presence of defects, never their absence.
3. Testing applies all through the software life cycle and is not an end of- cycle activity.
4. Understand the reason behind the test. 5. Test the tests first.

Testing Objective:

The main objective of testing is

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error

TYPE OF TESTING APPROACH

Verification and Validation approach:

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.

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

Levels of Testing:

Unit level Procedure:

Unit is the smallest part of a software system which is testable it may include code files, classes and methods which can be tested individual for correc tness. Unit is a process of validating such small building block of a complex system, much before testing an integrated large module or the system as a whole. Driver and/or stub software must be developed for each unit test A driver is nothing more than a "main program" that accepts test case data, passes such data to the component, and prints relevant results. Stubs serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface.

Integration Testing:

Integration is defined as a set of integration amoung component. Testing the interactions between the module and interactions with other system externally is called Integration Testing.

Type of Integration Testing

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

TOP-DOWN INTEGRATION:

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module.

BOTTOM-UP INTEGRATION:

Bottom-up integration testing, begins construction and testing with atomic modules. Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

REGRESSION TESTING:

Each time a new module is added as part of integration testing, the software changes. These changes may cause problems. In the context of an integration test strategy, regression testing is the re-execution of some subset of tests that have already been conducted. Regression testing is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors. Regression testing may be conducted manually, by re-executing a subset of all test cases.

SMOKE TESTING:

Smoke testing is an integration testing approach that is commonly used when “shrink wrapped” software products are being developed, allowing the software team to assess its project on a frequent basis. The smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a "build. " A build includes all data files, libraries, reusable modules, and engineered components.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function.
3. The build is integrated with other builds and the entire product is smoke tested daily. The integration approach may be top down or bottom up.

System Testing:

Main Objective	System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.
When to perform system testing	After the integration testing
Who are going to perform	Development team and user
Method	Problem or configuration management
Input	Requirement document, test plan, system test plan
Output	Test report
Tool	Tool Automated tool

INTEGRATION TECHNIQUE

Common Integration Technique:
In this section, we review some of the common approaches in the integration testing of object- oriented programs.

State Based Testing:
State-based testing techniques rely on the construction of a finite-state machine (FSM) or state-transition diagram to **represent the change of states of the program under test**. For integration testing, the construction of global FSM may become unmanageable and subject to the state-explosion problem. Conventional techniques for concurrent programs treat a program as static set of communicating components and model it as deterministic or non-deterministic FSMs communicating with one another. They systematically arrange the components into an FSM hierarchy by reducing composite FSMs at each level by means of abstraction, and classifying interaction statements as local and global points [8]. Alternatively, they may prune out unnecessary or infeasible state transitions, such as by employing interface processes. Some methods even store

the removed details in the edges. The resultant graphs often consist of much fewer nodes and edges than the flattened composition of all FSMs, so that it will be feasible to traverse all nodes and edges. In this way, the state-explosion problem can be alleviated.

Event Based Testing:

Instead of using the state-based approach, the synchronization sequence for a concurrent program can also be **viewed as relationships between pairs of synchronization events**. A merit of these approaches is that they support the analysis of event sequencing requirements without having to cater for the states of the program or components.

Testing Against Formal Specification:

A lot of research has been done using formal specifications for the testing of object oriented programs at the class level. For example, Doong and Frankl have proposed to test behavioral equivalence of two objects in a class by applying algebraic specification techniques. However, relative little work has been conducted for integration testing.

Contract is a formal language for specifying the behavioral dependencies and interactions among objects of different classes. Such behavioral properties are defined using "message- passing rules". Testing procedures have been defined for individual mp-rules as well as composite mp-rule. Hence, they have implemented two automatic testing tools for integration testing using Arity/Prolog.

Introducing Automated Testing and Test Complete Automated Testing is the automatic execution of software testing by a special program with little or no human interaction. Automated execution guarantees that no test action will be skipped ; it relieves tester of same boring step over and over. Developers write unit test cases in the form of little programs, typically in a framework such as JUnit

Test Complete provide special feature for automating test action, creating test, defining baseline data, running test and logging test result.

For example; "Recording Test" feature that create test visually just need to start recording, perform all the needed action against the tested application and test complete will automatically convert all the recorded action to a test.

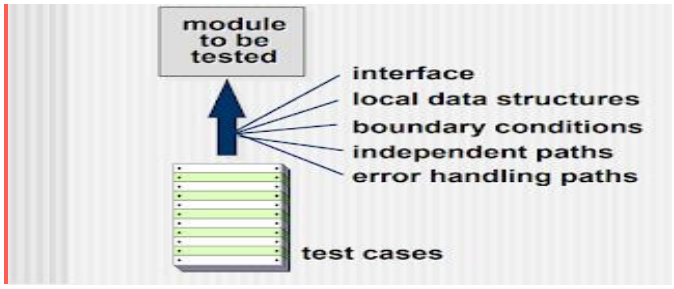
Test strategies for Conventional Software

- 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.

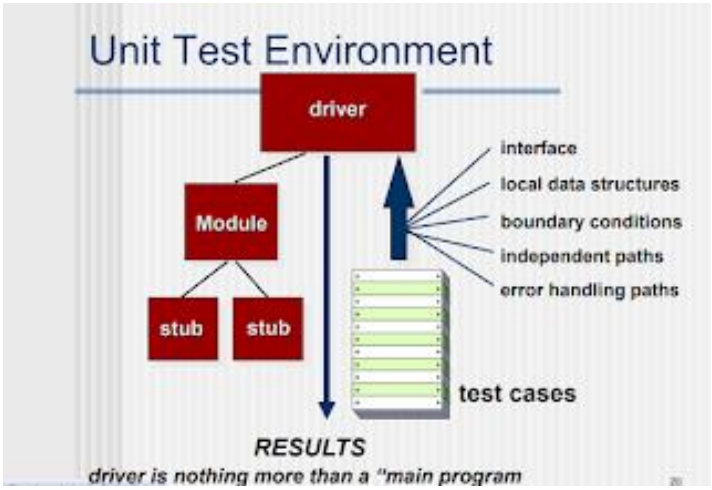
Unit Test :

- **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.
- *Unit testing is simplified when a component with high cohesion is designed.*



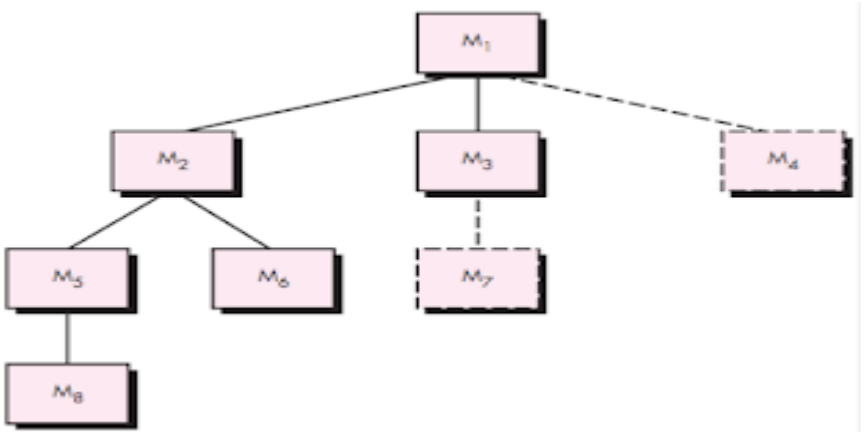
Integration Testing :

- 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.***



Integration Testing :

The integration process is performed in a series of five steps

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

Problem encountered during top-down integration testing.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise.

The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels.

Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure.

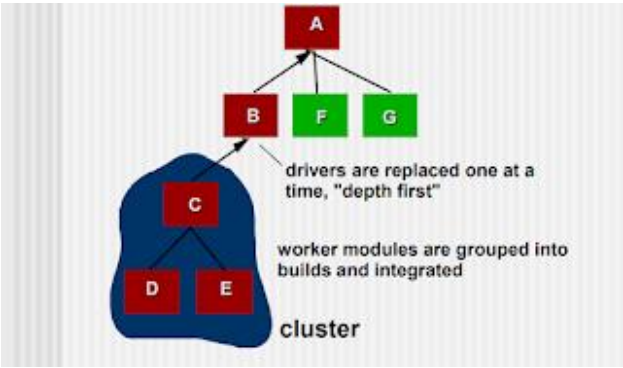
As a tester, you are left with three choices:

(1) delay many tests until stubs are replaced with actual modules,

(2) develop stubs that perform limited functions that simulate the actual module, or

(3) integrate the software from the bottom of the hierarchy upward.

Bottom-Up Integration



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 Step:**
- 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 Example**
- For Example in a project there are five modules like login, view user, user detail page, new user creation, and task creation etc. So in this five modules first of all developer perform the smoke testing by executing all the major functionality of modules like user is able to login or not with valid login credentials and after login new user can created or not, and user that is created viewed or not. So it is obvious that this is the smoke testing always done by developing team before submitting (releasing) the build to the testing team.

- Now once the build is released to the testing team than the testing team has to check whether to accept or reject the build by testing the major functionality of the build. So this is the smoke test done by testers
- **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**

1. **Black Box Testing** is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester

2. **White Box Testing** is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

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.
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 software.	It is the logic testing of the 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**

Validation Testing

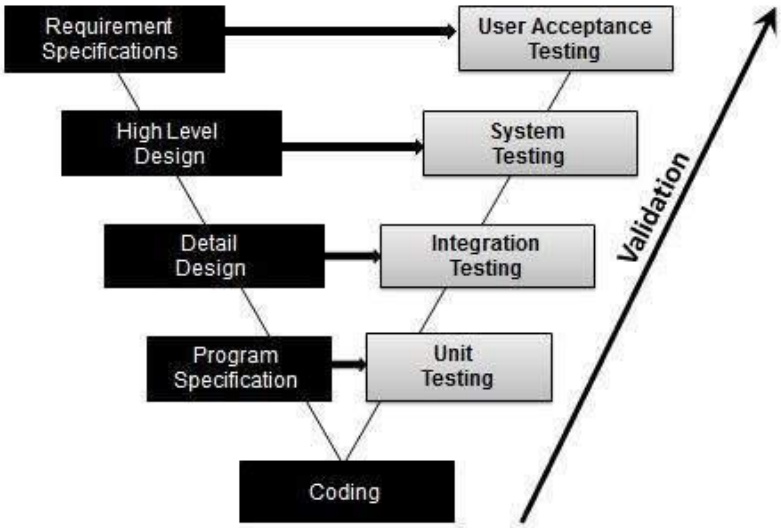
The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

It answers to the question, Are we building the right product?

Validation Testing - Workflow:

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.



Activities:

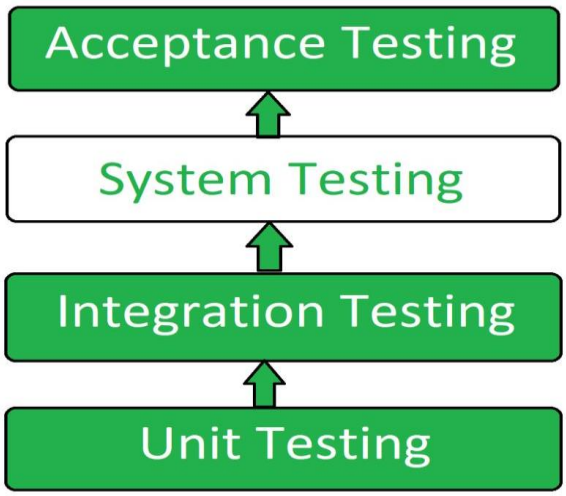
- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing

System Testing

System Testing is basically performed by a testing team that is independent of the development team that helps to test the quality of the system impartial. It has both functional and non-functional testing.

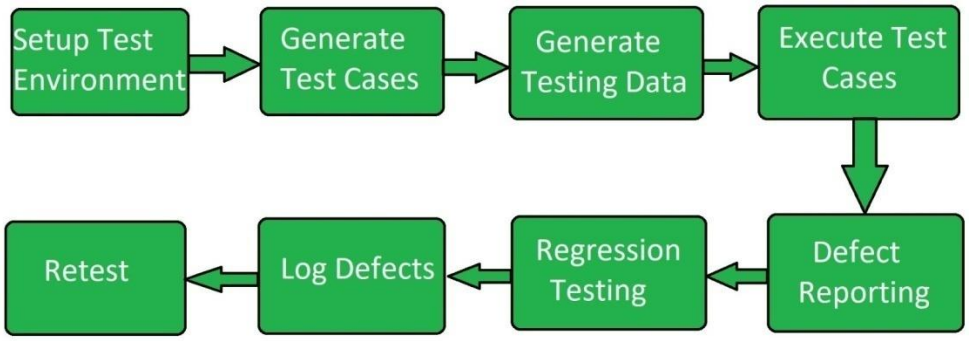
System Testing is a black-box testing.

System Testing is performed after the integration testing and before the acceptance testing.



System Testing Process:
System Testing is performed in the following steps:

- **Test Environment Setup:**
  Create testing environment for the better quality testing.
- **Create Test Case:**
  Generate test case for the testing process.
- **Create Test Data:**
  Generate the data that is to be tested.
- **Execute Test Case:**
  After the generation of the test case and the test data, test cases are executed.
- **Defect Reporting:**
  Defects in the system are detected.
- **Regression Testing:**
  It is carried out to test the side effects of the testing process.
- **Log Defects:**
  Defects are fixed in this step.
- **Retest:**
  If the test is not successful then again test is performed.



Types of System Testing:

- **Performance Testing:**
  Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application.
- **Load Testing:**
  Load Testing is a type of software Testing which is carried out to determine the behavior of a system or software product under extreme load.
- **Stress Testing:**
  Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads.
- **Scalability Testing:**
  Scalability Testing is a type of software testing which is carried out to check the

performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

The art of debugging

Introduction:

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.

Debugging Process: Steps involved in debugging are:

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

Debugging Strategies:

1. Study the system for the larger duration in order to understand the system. It helps debugger to construct different representations of systems to be debugging depends on the need. Study of the system is also done actively to find recent changes made to the software.
2. Backwards analysis of the problem which involves tracing the program backward from the location of failure message in order to identify the region of faulty code. A detailed study of the region is conducting 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 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.

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:

- Radare2
- WinDbg
- 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 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.

Software Quality
Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc., for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

SOFTWARE QUALITY ATTRIBUTES APPROACH

This approach to software quality is best exemplified by fixed quality models, such as ISO/IEC 25010:2011. This standard describes a hierarchy of eight quality characteristics, each composed of sub-characteristics:

1. Functional suitability
2. Reliability
3. Operability
4. Performance efficiency
5. Security
6. Compatibility
7. Maintainability
8. Transferability

Additionally, the standard defines a quality-in-use model composed of five characteristics:

1. Effectiveness
2. Efficiency
3. Satisfaction
4. Safety
5. Usability

A fixed software quality model is often helpful for considering an overall understanding of software quality. In practice, the relative importance of particular software characteristics typically depends on software domain, product type, and intended usage. Thus, software characteristics should be defined for, and used to guide the development of, each product.

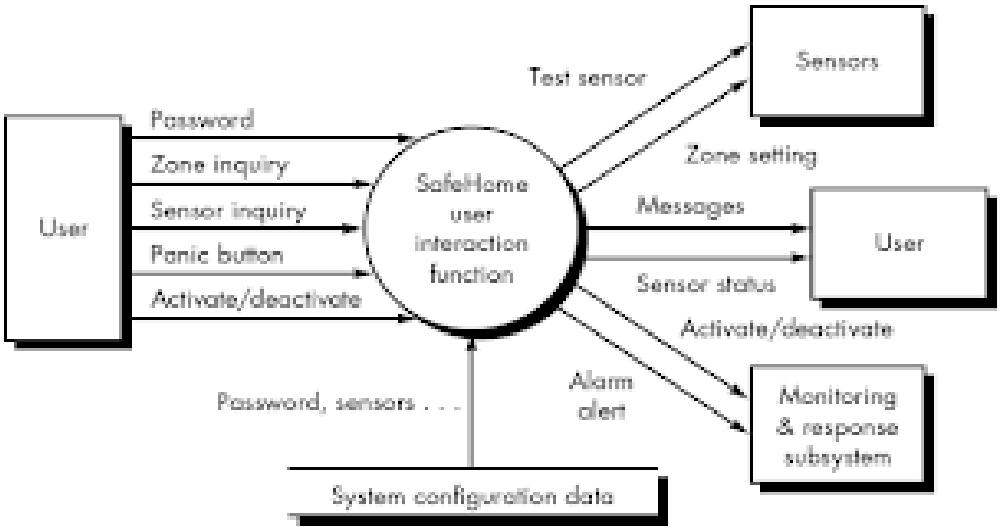
Metrics for Analysis model

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.

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics derived for project application for use in this context. These metrics examine the analysis model with the intent of predicting the "size" of the resultant system. It is likely that size and design complexity will be directly correlated.

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. To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in figure. Referring to the figure, a data flow diagram for a function within the SafeHome software is represented. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The function displays a series of prompting messages and sends appropriate control signals to various components of the security system.



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

Three user inputs—password, panic button, and activate/deactivate—are shown in the figure along with two inquires—zone inquiry and sensor inquiry. One file (system configuration file) is shown. Two user outputs (messages and sensor status) and four external interfaces (test sensor, zone setting, activate/deact

			Weighting Factor				
Measurement parameter	**Count**		**Simple**	**Average**	**Complex**		
Number of user inputs	3	×	3	4	6	=	9
Number of user outputs	2	×	4	5	7	=	8
Number of user inquiries	2	×	3	4	6	=	6
Number of files	1	×	7	10	15	=	7
Number of external interfaces	4	×	5	7	10	=	20
Count total							50

The count total shown in Figure 19.4 must be adjusted using Equation :

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." For the purposes of this example, we assume that (Fi) is 46 (a moderately complex product). Therefore,

FP = 50 [0.65 + (0.01 46)] = 56

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the SafeHome user interaction function. Assume that past data indicates that one FP translates into 60 lines of code (an objectoriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the analysis model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during analysis and design reviews and four errors per function point during unit and integration testing. These data can help software engineers assess the completeness of their review and testing activities.

The Bang Metric

Like the function point metric, the bang metric can be used to develop an indication of the size of the software to be implemented as a consequence of the analysis model. Developed by DeMarco, the bang metric is “an implementation independent indication of system size.” To compute the bang metric, the software engineer must first evaluate a set of primitives—elements of the analysis model that are not further subdivided at the analysis level. Primitives are determined by evaluating the analysis model and developing counts for the following forms:

Functional primitives (FuP). The number of transformations (bubbles) that appear at the lowest level of a data flow diagram.

Data elements (DE). The number of attributes of a data object, data elements are not composite data and appear within the data dictionary.

Objects (OB). The number of data objects .

Relationships (RE). The number of connections between data objects .

States (ST). The number of user observable states in the state transition diagram.

Transitions (TR). The number of state transitions in the state transition diagram.

Metrics for Design Model:

Metrics simply measures quantitative assessment that focuses on countable values most commonly used for comparing and tracking performance of system. Metrics are used in different scenarios like analyzing model, design model, source code, testing, and maintenance. Metrics for design modeling allows developers or software engineers to evaluate or estimate quality of design and include various architecture and component-level designs.

Metrics by Glass and Card :
In designing a product, it is very important to have efficient management of complexity. Complexity itself means very difficult to understand. We know that systems are generally complex as they have many interconnected components that make it difficult to understand. Glass and Card are two scientists who have suggested three design complexity measures. These are given below :

1. **Structural Complexity:**

   Structural complexity depends upon fan-out for modules. It can be defined as :

   $$S(k) = f^2_{out}(k)$$

   Where $\mathbf{f_{out}}$ represents fanout for module k (fan-out means number of modules that are subordinating module k).

2. **Data Complexity:**

   Data complexity is complexity within interface of internal module. It is size and intricacy of data. For some module k, it can be defined as :

   $$D(k) = tot\_var(k) / [f_{out}(k)+1]$$

   Where *tot_var* is total number of input and output variables going to and coming out of module.

3. **System Complexity:**

System complexity is combination of structural and data complexity. It can be denoted as:
Sy(k) = S(k)+D(k)

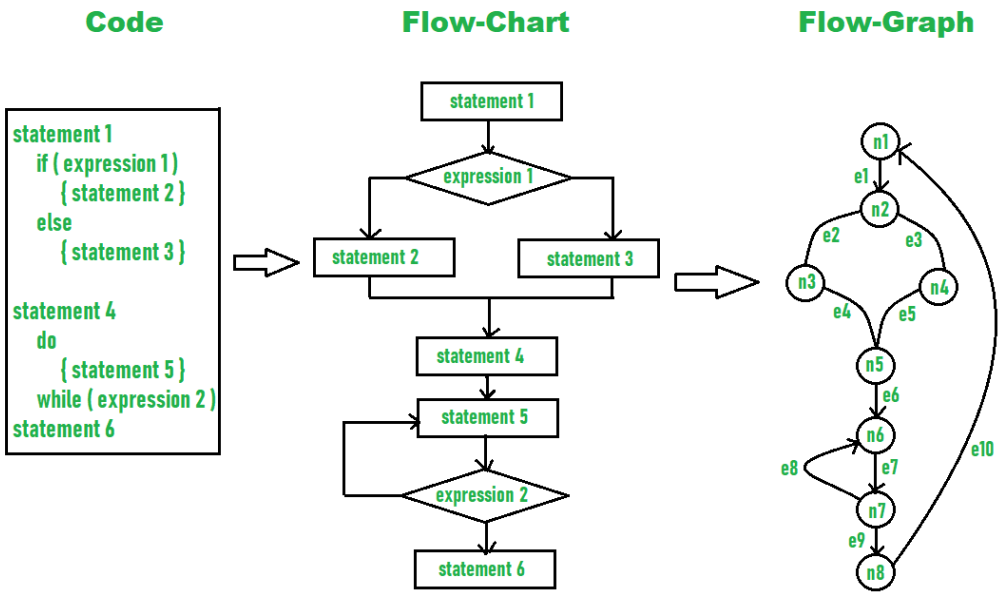
When structural, data, and system complexity get increased, overall architectural complexity also gets increased.

Complexity metrics:

Complexity metrics are used to measure complexity of overall software. The computation if complexity metrics can be done with help of a flow graph. It is sometimes called cyclomatic complexity. The cyclomatic complexity is a useful metric to indicate complexity of software system. Without use of complexity metrics, it is very difficult and time-consuming to determine complexity in designing products where risk cost emanates. Even continuous complexity analysis makes it difficult for project team and management to solve problem. Measuring Software complexity leads to improve code quality, increase productivity, meet architectural standards, reduce overall cost, increases robustness, etc. To calculate cyclomatic complexity, following equation is used:
Cyclomatic complexity= E - N + 2

Where, E is total number of edges and N is total number of nodes.

Example
In diagram given below, you can see number of edges and number of nodes.



So, the Cyclomatic complexity can be calculated as Given,

E = 10,
N = 8
So,

Cyclomatic complexity

= E - N + 2
= 10 – 8 + 2
= 4

Source code metrics

These are measurements of the source code that make up all your software. Source code is the fundamental building block of which your software is made, so measuring it is key to making sure your code is high-caliber. (Not to mention there is almost always room for improvement.) Look closely enough at even your best source code, and you might spot a few areas that you can optimize for even better performance.

When measuring source code quality make sure you're looking at the number of lines of code you have, which will ensure that you have the appropriate amount of code and it's no more complex than it needs to be. Another thing to track is how compliant each line of code is with the programming languages' standard usage rules. Equally important is to track the percentage of comments within the code, which will tell you how much maintenance the program will require. The less comments, the more problems when you decide to change or upgrade. Other things to include in your measurements is code duplications and unit test coverage, which will tell you how smoothly your product will run.

Development metrics

These metrics measure the custom software development process itself. Gather development metrics to look for ways to make your operations more efficient and reduce incidents of software errors.

Measuring number of defects within the code and time to fix them tells you a lot about the development process itself. Start by tallying up the number of defects that appear in the code and note the time it takes to fix them. If any defects have to be fixed multiple time then there might be a misunderstanding of requirements or a skills gap – which is important to address as soon as possible.

Testing metrics

These metrics help you evaluate how functional your product is. There are two major testing metrics. One of them is “test coverage” that collects data about which parts of the software program are executed when it runs a test. The second part is a test of the testing itself. It’s called “defect removal efficiency,” and it checks your success rate for spotting and removing defects.

The more you measure, the more you know about your software product, the more likely you are able to improve it. Automating the measurement process is the best way to measure software quality – it’s not the easiest thing, or the cheapest, but it’ll save you tons of cost down the line.

Metrics for Software Maintenance

When development of a software product is complete and it is released to the market, it enters the maintenance phase of its life cycle. During this phase the defect arrivals by time interval and customer problem calls (which may or may not be defects) by time interval are the de facto metrics. However, the number of defect or problem arrivals is largely determined by the development process before the maintenance phase. Not much can be done to alter the quality of the product during this phase. Therefore, these two de facto metrics, although important, do not reflect the quality of software maintenance. What can be done during the maintenance phase is to fix the defects as soon as possible and with excellent fix quality. Such actions, although still not able to improve the defect rate of the product, can improve customer satisfaction to a large extent. The following metrics are therefore very important:

- 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 a workload statement for software maintenance. It is related to both 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. Another metric to

manage the backlog of open, unresolved, problems is the backlog management index (BMI).

$$\text{BMI} = \frac{\text{Number of problems closed during the month}}{\text{Number of problem arrivals during the month}} \times 100\%$$

Fix Response Time and Fix Responsiveness

For many software development organizations, guidelines are established on the time limit within which the fixes should be available for the reported defects. Usually the criteria are set in accordance with the severity of the problems. For the critical situations in which the customers' businesses are at risk due to defects in the software product, software developers or the software change teams work around the clock to fix the problems. For less severe defects for which circumventions are available, the required fix response time is more relaxed. The fix response time metric is usually calculated as follows for all problems as well as by severity level:

Percent Delinquent Fixes

The mean (or median) response time metric is a central tendency measure. A more sensitive metric is the percentage of delinquent fixes. For each fix, if the turnaround time greatly exceeds the required response time, then it is classified as delinquent:

$$\text{Percent delinquent fixes} = \frac{\text{Number of fixes that exceeded the response time criteria by severity level}}{\text{Number of fixes delivered in a specified time}} \times 100\%$$

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. From the customer's perspective, it is bad enough to encounter functional defects when running a business on the software. It is even worse if the fixes turn out to be defective. 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.