

Unit-I

Introduction to Software Engineering

Computer software, or simply **software**, is a collection of data or computer instructions that tell the computer how to work.

Software is a collection of computer programs, procedures, rules, and associated documentation and data. [IEEE]

Engineering is the creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behaviour under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property. [American Engineers' Council for Professional Development (ABET)].

[Engineering is the branch of science and technology concerned with the design, building, and use of engines, machines, and structures.]

Software Engineering

Software engineering is the systematic application of engineering approaches to the development of software. Software engineering is a sub-field of computing science.

Definitions of software engineering:

“The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software”—The Bureau of Labor Statistics—IEEE *Systems and software engineering – Vocabulary*.

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software”—IEEE *Standard Glossary of Software Engineering Terminology*

“The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines” — Fritz Bauer, 1968

“A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements” — Stephen Schach

“Software engineering is an engineering discipline which is concerned with all aspects of software production” — Ian Sommerville

“A branch of computer science that deals with the design, implementation, and maintenance of complex computer programs”—Merriam-Webster

Software Characteristics

Software is defined as collection of computer programs, procedures, rules and data. Different individuals judge software on different basis. This is because they are involved with the software in different ways. For example, users want the software to perform according to their requirements. Similarly, developers involved in designing, coding, and maintenance of the software evaluate the software by looking at its internal characteristics, before delivering it to the user. Software characteristics are classified into six major components:

Functionality. It refers to the degree of performance of the software against its intended purpose. Here one considers the suitability, accuracy, interoperability, compliance and security of the software.

Reliability. It refers to the ability of the software to provide desired functionality under the given conditions. Required functions are recoverability, fault tolerance and maturity.

Usability. It refers to the extent to which the software can be used with ease. The amount of effort or time required to learn how to use the software. Functions required are understandability, learnability and operability.

Efficiency. It refers to the ability of the software to use system resources in the most effective and efficient manner. Functions required are efficiency in time and in resources.

Maintainability. It refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors. Required functions are testability, stability, changeability and operability.

Portability. It is the ease with which software developers can relaunch software from one platform to another, without (or with minimum) changes. In simple terms, software must be made in way that it should be platform independent. In other words, it is a set of attribute that bear on the ability of software to be transferred from one environment to another, without or minimum changes. Required functions are adaptability, installability and replaceability.

Remarks: Software does not wear out. It is not manufactured. It is flexible. Components are reusable.

Software versus Hardware

Software possesses no mass, no volume, and no colour, which makes it a non-degradable entity over a long period. It does not wear out or get tired.

Software Components

What is software component?

A software unit of functionality that manages a single abstraction.

A software component is basically a software unit with a well-defined interface and explicitly specified dependencies. A software component can be as small as a block of reusable code, or it can be as big as an entire application.

A system element offering a predefined service and able to communicate with other components.

Reusable software part that can be easily plugged together with other components to form a software application.

Components of software: Programs, Documentation and Operating Procedures

Program and Software. Program is a subset of SW

Characteristics of software. It does not wear out. It is not manufactured. Its components are reusable. It is flexible.

Why Software Engineering?

Software Engineering is required due to the following reasons:

To manage large software. As the measure of programming become extensive engineering has to step to give it a scientific process.

For more scalability/adaptability. If the software procedure were not based on scientific and engineering ideas, it would be simpler to re-create new software than to scale an existing one.

Cost Management. As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the proper process is not adapted.

To manage the dynamic nature of software. The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one.

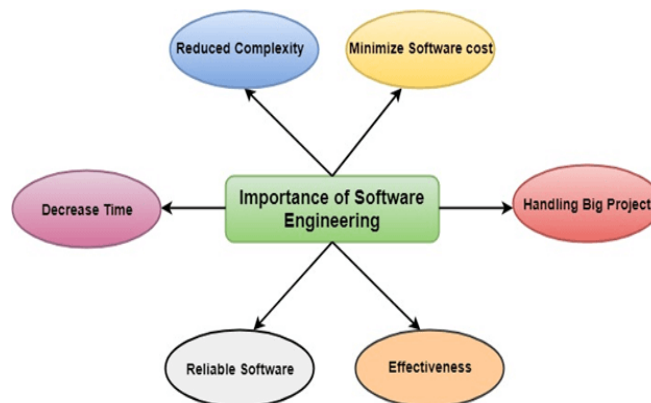
For better quality management. Better procedure of software development provides a better and quality software product.

Importance of Software Engineering

Reduces complexity: Big software is always complicated and challenging to progress. Software engineering has a great solution to reduce the complication of any project. Software engineering divides big problems into various small issues. And then start solving each small issue one by one. All these small problems are solved independently to each other.

To minimize software cost: Software needs a lot of hardwork and software engineers are highly paid experts. A lot of manpower is required to develop software with a large number of codes. But in software engineering, programmers project everything and decrease all those things that are not needed. In turn, the cost for software productions becomes less as compared to any software that does not use software engineering method.

To decrease time: Anything that is not made according to the project always wastes time. And if you are making great software, then you may need to run many codes to get the definitive running code. This is a very time-consuming procedure, and if it is not well handled, then this can take a lot of time. So if you are making your software according to the software engineering method, then it will decrease a lot of time.



Handling big projects: Big projects are not done in a couple of days, and they need lots of patience, planning, and management. And to invest six and seven months of any company, it requires heaps of planning, direction, testing, and maintenance. No one can say that he has given four months of a company to the task, and the project is still in its first stage. Because the company has provided many resources to the plan and it should be completed. So to handle a big project without any problem, the company has to go for a software engineering method.

Reliable software: Software should be secure, means if you have delivered the software, then it should work for at least its given time or subscription. And if any bugs come in the software, the company is responsible for solving all these bugs. Because in software engineering, testing and maintenance are given, so there is no worry of its reliability.

Effectiveness: Effectiveness comes if anything has made according to the standards. Software standards are the big target of companies to make it more effective. So Software becomes more effective in the act with the help of software engineering.

Characteristics of a good software engineer

Exposure to systematic methods, i.e., familiarity with software engineering principles.

Good technical knowledge of the project range (Domain knowledge).

Good programming abilities.

Good communication skills. These skills comprise of oral, written, and interpersonal skills.

High motivation.

Sound knowledge of fundamentals of computer science.

Intelligence.

Ability to work in a team

Discipline, etc.

Software Applications

For sake of convenience, applications of software are grouped in eight areas.

System Software. Infrastructure software comes under this category like compilers, operating systems, editors, drivers, etc. Basically, system software is a collection of programs to provide service to other programs.

Real Time Software. These softwares are used to monitor, control and analyse real world events as they occur. An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forecast the weather.

Embedded Software. This type of software is placed in "Read-Only-Memory(ROM)" of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software.

Business Software. Software finds its largest application in this area. The software designed to process business applications is called business software. Business software could be a payroll, file monitoring system, employee management, account management. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software.

Personal Computer Software. The software used in personal computers come under this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games, etc. This is a very upcoming area and many big organisations are concentrating their efforts here due to large customer base.

Artificial Intelligence Software. AI software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Examples are expert systems, artificial neural network, signal processing software, etc.

Web based software. The software related to web applications come under this category. Examples are CGI, HTML, Java, Perl, DHTML, etc.

[CGI's full form is *Common Gateway Interface*. It offers a standard protocol for web servers to execute programs that execute like console applications running on a server that generates web pages dynamically. Such programs are known as CGI scripts or simply as CGIs. It is most commonly used to refer to 3D computer graphics. Used to create scenes and 3D effects in TVs and films. Avatar, Toy Story (by PIXAR, Steve Jobs), etc are where CGIs are used extensively.

HTML's full form is *Hyper Text Markup Language*. Computers use language to communicate just like people do. Computers don't communicate like human does not mean they don't communicate. They don't think like human does not mean they don't think. The way computers communicate to the internet is through a computer language called HTML. It is the standard markup language for creating web pages and applications. Most widely used language on web to develop web pages. Hypertext is a process of linking objects to each other. When an object is clicked the linked object could be seen. Markup language and Programming language are not the same. DHTML's full form is *Dynamic HTML*. It is the art of combining HTML, Javascript, DOM and CSS. It allows documents to be animated. It is not a language or a web standard. Full form of DOM is Document Object Model. *CSS: Cascading Style Sheets*. It defines how to display HTML elements. *Javascript* is the most popular scripting language on the internet and it works in all major browsers.

Engineering and Scientific Software. Scientific and engineering application software are grouped in this category. Huge computing is normally required to process data. Examples are CAD/CAM package, SPSS, MATLAB, Engineering Pro, Circuit analysers etc.

[CAM: *Computer-aided manufacturing* is the use of software to control machine tools and related ones in the manufacturing of workpieces. This is not the only definition for CAM, but it is the most common one; CAM may also refer to the use of a computer to assist in all operations of a manufacturing plant, including planning, management, transportation and storage. Its primary purpose is to create a faster production process and components and tooling with more precise dimensions and material consistency, which in some cases, uses only the required amount of raw material (thus minimizing waste), while simultaneously reducing energy consumption. CAM is now a system used in schools and lower educational purposes. CAM is a subsequent computer-aided process after *computer-aided design*(CAD) and sometimes *computer-aided engineering*(CAE), as the model generated in CAD and verified in CAE can be input into CAM software, which then controls the machine tool. CAM is used in many schools alongside CAD: *Computer-Aided Design*.]

Software Engineering Principles

- (i) Software metrics and measurement. Software metrics are used to characterise different aspects of software process or software products. Examples: Process metrics include productivity, quality, failure rate, efficiency etc. Product metrics are size, reliability, complexity, functionality etc.
- (ii) Software monitoring and control.

(iii) SDLC: Waterfall model, Prototyping model, Incremental model, Iterative enhancement model, Spiral model.

Waterfall model: Requirements analysis and specifications, Design, Implementation and Unit Testing, Integration and System Testing, Operation and Maintenance.

We discussed these five models of SDLC in detail (all the different phases involved in the development) and mentioning its advantages and disadvantages in all the cases.

Unit-II. Software Requirements Specification.

- (i) Definition of Requirements Engineering.
- (ii) The various process steps of requirement engineering.

The process consists of four steps:

- (a) Requirements elicitation
- (b) Requirements analysis
- (c) Requirements documentation (SRS document, extremely important, it clearly laid out the captured requirements of the clients, the outcome of studying the present chapter).
- (d) Requirements review/validation

(iii) Types of requirements: Functional (describes what the software has to do/not to do) and non-functional requirements (specifications of desired performance, availability, reliability, usability and flexibility). Non-functional requirements for developers are maintainability, portability and testability.

(iv) User and system requirements: Both are parts of SRS document.

User requirements are written for the users and include functional and non-functional requirements. It describes external behaviour of the system with some constraints and parameters without the use of any technical jargon. System requirements are derived from user requirements. Used as input to the designers to prepare SDD.

(v) Feasibility studies.

Are the project's cost and schedule assumptions realistic?

Is the business model realistic?

Is there any market for the product?

(vi) We studied the various process steps in detail.

(a) **Requirements elicitation techniques:** Interviews, Brainstorming sessions, Facilitated Application Specification Technique, Quality Function Deployment, The Use Case Approach.

(b) **Requirements analysis:** Steps involved are: Draw the context diagrams, Development of a prototype(optional), Model the requirements, Finalise the requirements.

(c) **Requirements documentation.** Also called Software Requirements Specifications (SRS). *Nature of the SRS:* Functionality, External interfaces, Performance, Attributes, Design constraints imposed on an implementation.

Characteristics of a good SRS: SRS should be correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, traceable.

(d) **Requirements Validation:** After the completion of SRS document, we may like to check the document for completeness and consistency, conformance to standards, requirements conflict, technical errors, ambiguous requirements. The objective of requirements validation is to certify that the SRS document is an acceptable document of the system to be implemented.

Requirements validation techniques: Requirements reviews, Prototyping.

Unit-III. System Design.

1. What is design?

Highly significant phase in the software development where the designer plans "how" a software should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain.

Design involves transformation of ideas into detailed implementation descriptions, with the goal of satisfying the software requirements (as specified in SRS document).

Why is design important?

Objectives of design:

Conceptual and Technical Designs. Designers are intermediators between the customers and the system builders. Designers prepare conceptual design that tells the customers exactly what the system will do. Once this is approved by the customers, the conceptual design is translated into a much more detailed document, the technical design, that allows system builders to understand the actual hardware and software needed to solve the customer's problem. Conceptual tells what the system will do. Technical design tells how the system is to work.

- 2. **Software Design Principles:** Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

Software design principles are: *Problem partitioning, Abstraction, Modularity, Top down and Bottom up strategy*

2.1 Problem Partitioning:

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning:

Software is easy to understand

Software becomes simple

Software is easy to test

Software is easy to modify

Software is easy to maintain

Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity. Note that as the number of partition increases, cost of partition and complexity increases.

2.2 Abstraction: Elimination of the irrelevant and the amplification of the essentials.

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

Here, there are two common abstraction mechanisms. Functional Abstraction & Data Abstraction.

Functional Abstraction- A module is specified by the method it performs. The details of the algorithm to accomplish the functions are not visible to the user of the function. Functional abstraction forms the basis for Function oriented design approaches.

*Data Abstraction-*Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for Object Oriented design approaches.

2.3 Modularity:

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

Each module is a well-defined system that can be used with other applications.

Each module has single specified objectives.

Modules can be separately compiled and saved in the library.

Modules should be easier to use than to build.

Modules are simpler from outside than inside.

Advantages and Disadvantages of Modularity

We will discuss various advantage and disadvantage of Modularity.

Advantages of Modularity

There are several advantages of Modularity

It allows large programs to be written by several or different people

It encourages the creation of commonly used routines to be placed in the library and used by other programs.

It simplifies the overlay procedure of loading a large program into main storage.

It provides more checkpoints to measure progress.

It provides a framework for complete testing, more accessible to test

It produced the well designed and more readable program.

Disadvantages of Modularity

There are several disadvantages of Modularity

Execution time maybe, but not certainly, longer

Storage size perhaps, but is not certainly, increased

Compilation and loading time may be longer

Inter-module communication problems may be increased

More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done.

Modular Design: Modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of various parts of a system. We discuss a different section of modular design in detail:

Functional Independence: Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules. Independence is important because it makes

implementation more accessible and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality.

It is measured using two criteria:

- **Cohesion:** It measures the relative function strength of a module.
- **Coupling:** It measures the relative interdependence among modules.

Information hiding: The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others. In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

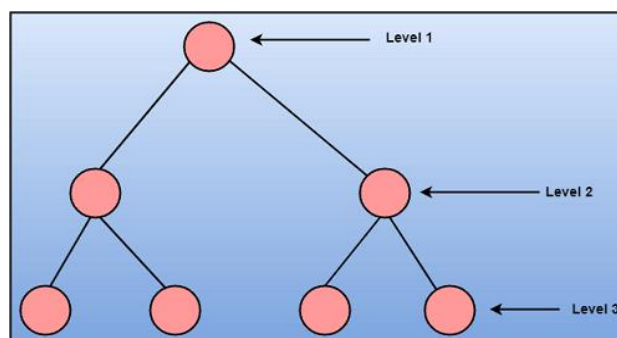
The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

2.4 Strategy of Design:

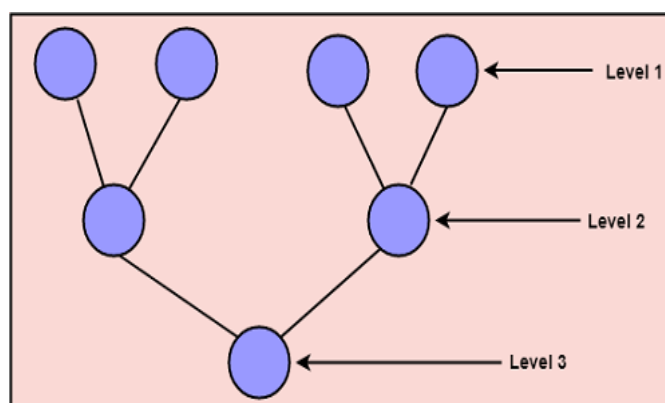
A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program. To design a system, there are two possible approaches:

- (a) Top-down Approach
- (b) Bottom-up Approach

Top-down Approach: This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.



Bottom-up Approach: A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.



Hybrid design:

3. Structured approach.

4. Function oriented design Vs Object oriented design.

5. Design specification:

6. Modularity: Simply dividing/chopping the software system into modules indefinitely wouldn't make things simpler. This idea give rise to the concept of modularity. Under modularity and over modularity should be avoided.

6.1 Module coupling: Measure of the degree of interdependence between modules. Uncoupled: no dependencies. Loosely coupled: some dependencies. Highly coupled: many dependencies.
A good design will have low coupling.

Types of coupling in order from lowest coupling (best) to highest coupling (worst):

Data coupling, Stamp coupling, Control coupling, External coupling, Common coupling, Content coupling.

6.2 Module cohesion (strength of relations within modules): Measure of the degree to which the elements of a module are functionally related. A good design will have maximum interaction within a module.

Types of cohesion in order from highest cohesion (best) to lowest cohesion (worst):

Functional cohesion, Sequential cohesion, Communicational cohesion, Procedural cohesion, Temporal cohesion, Logical cohesion, Coincidental cohesion.

Relationship between Cohesion and Coupling. A software engineer must design the modules with the goal of high cohesion and low coupling.

Design notations: For a function oriented design, the design can be represented graphically or mathematically by the following: Data flow diagrams (DFDs), Data dictionaries, Structure charts, Pseudocode.

7. Overview of SA/SD Methodology

8. Structured Analysis and Design Tools:

Activities helping the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.

Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.

Let us see few analysis and design tools used by software designers:

I. Data flow diagrams

Data Flow Diagram (DFD) is a graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow, and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. It does not contain any control or branch elements.

Types of DFD

Data Flow Diagrams are either Logical or Physical.

Logical DFD - This type of DFD concentrates on the system process, and flow of data in the system. For example in a banking software system, how data is moved between different entities.

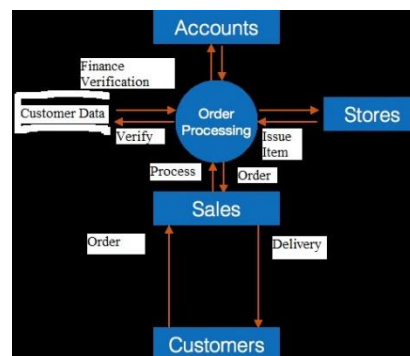
Physical DFD - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

Levels of DFD

Level 0 - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



Level 1 - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.



Level 2 - At this level, DFD shows how data flows inside the modules mentioned in Level 1. Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

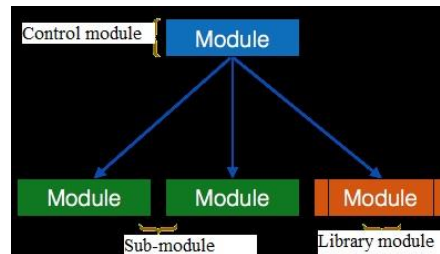
II. Structure charts

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.

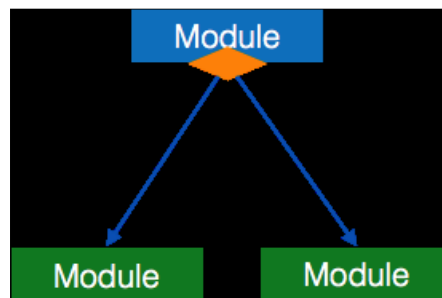
Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

Here are the symbols used in construction of structure charts -

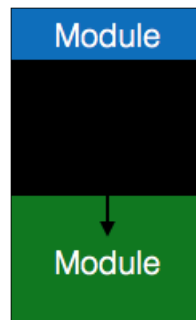
Module - It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and invocable from any module.



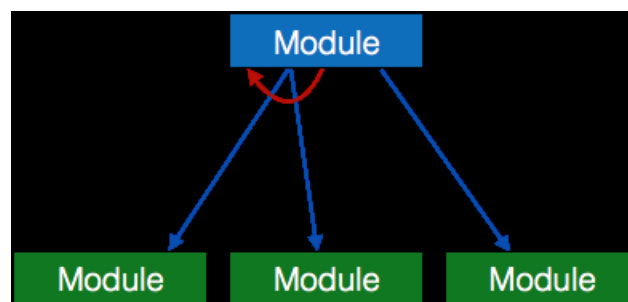
Condition - It is represented by small diamond at base of the module. It depicts that control module can select any of sub-routine based on some condition.



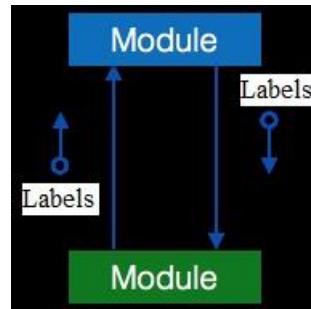
Jump - An arrow is shown pointing inside the module to depict that the control will jump in the middle of the sub-module.



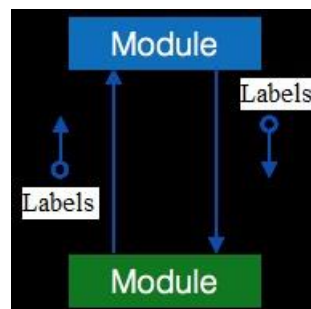
Loop - A curved arrow represents loop in the module. All sub-modules covered by loop repeat execution of module.



Data flow - A directed arrow with empty circle at the end represents data flow.



Control flow - A directed arrow with filled circle at the end represents control flow.



Unit-IV Software Project Management

The job pattern of an IT company engaged in software development can be seen split in two parts: Software Creation, Software Project Management.

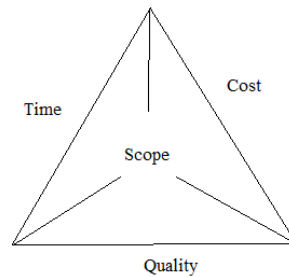
A project is well-defined task, which is a collection of several operations done in order to achieve a goal (for example, software development and delivery).

Software Project. A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

Project Success: In order to conduct a successful software project, we must understand: **Scope of work to be done**, **the risk to be incurred**, the resources required, **the task to be accomplished**, **the cost to be expended**, **the schedule to be followed**.

Importance of/Need for software project management.

Software is said to be an intangible product. Software development is a kind of all new stream in world business and there's very little experience in building software products. Most software products are tailor made to fit client's requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.



The image above shows triple constraints for software projects.

There are three needs for software project management. These are: Time, Cost, Quality

It is an essential part of software organization to deliver quality product, keeping the cost within client's budget and deliver the project as per schedule. There are several factors, both internal and external, which may impact these three factors. Any of the three factor can severely impact the other two. Therefore, software project management is essential to incorporate user requirements along with budget and time constraints.

Goals: Deliver the software to the customers at the agreed time. Keep overall costs within budget. Deliver software that meets the customer's expectations. Maintaining a happy and well-functioning development team.

Challenges: The product is intangible. Large software projects are often 'one-off' projects. Software processes are variable and organisation-specific.

Software Project Manager. A software project manager is a person who undertakes the responsibility of executing the software project. Software project manager is thoroughly aware of all the phases of SDLC that the software would go through. Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

A project manager closely monitors the development process, prepares and executes various plans, arranges necessary and adequate resources, maintains communication among all team members in order to address issues of cost, budget, resources, time, quality and customer satisfaction.

Let us see few responsibilities that a project manager shoulders -**Managing People:** Act as project leader, contact with stakeholders, managing human resources, setting up reporting hierarchy, etc. **Managing Project.** Defining and setting up project scope, managing project management activities, Monitoring progress and performance, Risk analysis at every phase, Take necessary step to avoid or come out of problems, Act as project spokesperson.

Software management activities(Responsibilities of software project managers): Project planning, Project resource management, Scope management, Project estimation, Risk management, Project scheduling, Configuration management, Proposal writing, Project reporting.

Project planning is the most important activity which we shall discuss in very detail.

Project planning: Software project planning is task, which is performed before the production of software actually starts. It is there for the software production but involves no concrete activity that has any direction connection with software production; rather it is a set of multiple processes, which facilitates software production. Provides roadmap for successful software engineering. Estimation of cost and development time are the key issues during project planning.

Activities in software project planning may include: Size estimation, Efforts estimation, Cost estimation, Development time, Resources requirements, Project scheduling

Size estimation: Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices and *Function points* vary according to the user or software requirement.

“A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declaration, and executable and non-executable

statements". **We shall discuss this in detail later**

Efforts estimation: The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers' experience, organization's historical data or software size can be converted into efforts by using some standard formulae.

Development time: Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months.

The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.

Cost estimation: This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider -Size of software, Software quality, Hardware, Additional software or tools, licenses etc., Skilled personnel with task-specific skills, Travel involved, Communication, Training and support

COCOMO Models: We shall discuss in detail later.

Resource requirements: In software Development, all the elements are referred to as resources for the project. It can be a human resource, productive tools, and libraries. Resource management includes: (i) create a project team and assign responsibilities to every team member (ii) developing a resource plan is derived from the project plan. (iii) adjustment of resources.

Project scheduling: The process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed. Estimation of calendar time, resources needed, budget. Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and arrange them keeping various factors in mind. They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.

For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

Scope management

It describes the scope of project and includes all the activities need to be undertaken in order to make a deliverable software product. Scope management is essential because it creates boundaries of the project by clearly defining what would be done in the project and what would not be done. This makes project to contain limited and quantifiable tasks, which can easily be documented and in turn avoids cost and time overrun.

Project estimation

This is not only about cost estimation because whenever we start to develop software, but we also figure out their size (line of code), efforts, time as well as cost. If we talk about the size, then Line of code depends upon user or software requirement.

If we talk about effort, we should know about the size of the software, because based on the size we can quickly estimate how big team required to produce the software. If we talk about time, when size and efforts are estimated, the time required to develop the software can easily determine.

And if we talk about cost, it includes all the elements such as:

Size of software, Quality, Hardware, Communication, Training, Additional Software and tools, Skilled manpower

Configuration management

Configuration management is about to control the changes in software like requirements, design, and development of the product. The Primary goal is to increase productivity with fewer errors.

Some reasons show the need for configuration management:

- Several people work on software that is continually update.
- Help to build coordination among suppliers.
- Changes in requirement, budget, schedule need to accommodate.
- Software should run on multiple systems.

Baseline. A phase of SDLC is assumed over if it baselined, i.e. baseline is a measurement that defines completeness of a phase. A phase is baselined when all activities pertaining to it are finished and well documented. If it was not the final phase, its output would be used in next immediate phase.

Configuration management is a discipline of organization administration, which takes care of occurrence of any change (process, requirement, technological, strategic etc.) after a phase is baselined. CM keeps check on any changes done in software.

Change Control

Change control is function of configuration management, which ensures that all changes made to software system are consistent and made as per organizational rules and regulations.

A change in the configuration of product goes through following steps -

- **Identification** - A change request arrives from either internal or external source. When change request is identified formally, it is properly documented.
- **Validation** - Validity of the change request is checked and its handling procedure is confirmed.
- **Analysis** - The impact of change request is analyzed in terms of schedule, cost and required efforts. Overall impact of the prospective change on system is analyzed.
- **Control** - If the prospective change either impacts too many entities in the system or it is unavoidable, it is mandatory to take approval of high authorities before change is incorporated into the system. It is decided if the change is worth incorporation or not. If it is not, change request is refused formally.
- **Execution** - If the previous phase determines to execute the change request, this phase take appropriate actions to execute the change, does a thorough revision if necessary.
- **Close request** - The change is verified for correct implementation and merging with the rest of the system. This newly incorporated change in the software is documented

People involved in Configuration Management: Project manager, Configuration manager, developers, users

Software metrics:

We discussed a little of this in the first unit.

Size measure is very simple, and important metric for software industry. It has many useful characteristics like: It is very easy to calculate, once the program is completed. It plays an important role and is one of the most important parameter for many software development models like cost and effort estimation. Productivity is also expressed in terms of size measure. Memory requirements can also be decided on the basis of size measure.

The principal size measures, which have got more attention than others, are: Lines of Code (LOC), Token count, Function count

Most commonly used metrics are **size**, complexity and reliability. It is the relationship of size with the cost and quality that makes size an important metric. We discussed a few size measures.

(a) Lines of code (LOC). It is one of the earliest and the simplest metric for calculating the size of a computer program. It is generally used in calculating and comparing the productivity of programmers. Productivity is measured as LOC / man-month. Among researchers, there is no general agreement what makes a line of code. Due to lack of standard and precise definition of LOC measure, different workers for the same program may obtain different counts. Further, it also gives an equal weightage to each line of code. But, in fact some statements of a program are more difficult to code and comprehend than others. Despite all this, this metric still continues to be popular and useful in software industry because of its simplicity.

The most important characteristic of this metric is its precise and standard definition. There is a general agreement among researchers that this measure should not include comment and blank lines because these are used only for internal documentation of the program. Their presence or absence does not affect the functionality, efficiency of the program. Some observers are also of the view that only executable statements should be included in the count, because these only support the functions of the program.

The predominant definition of LOC measure used today by various software personnel is:

"Any line of program text excluding comment or blank lines, regardless of the number of statements or parts of statements on the line, is considered a line of code (LOC). It excludes all lines containing program headers, declarations, and non-executable statements and includes only executable statements."

[Most common measure of size. LOC metric should not include comments or blank lines since these are really internal documentation and their presence or absence does not affect the functions of the program.

Disadvantage: Depends on language used. Even when using the same language, the size depends on how the lines are counted].

(b) Token Count. The drawback in LOC size measure of treating all lines alike can be solved by giving more weight to those lines, which are difficult to code and have more "stuff". One natural solution to this problem may be to count the basic symbols used in a line instead of lines themselves. These basic symbols are called "tokens". Such a scheme was used by Halstead in his theory of software science. In this theory, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these basic symbols. The basic measures are:

n_1 = count of unique operators

n_2 = count of unique operands

N_1 = count of total occurrences of operators

N_2 = count of total occurrences of operands

An operator can be defined as a symbol or keyword, which specifies an action. Operators consist of arithmetic, relational symbols, punctuation marks, special symbols (like braces, =), reserve-word/keywords (like WHILE, DO, READ) and function names like printf (), scanf () etc. A token, which receives the action and is used to represent the data, is called an operand. Operands include variables, constants and even labels.

In terms of the total tokens used, the size of the program can be expressed as:

$$N = N_1 + N_2$$

At present, there is no general agreement among researchers on counting rules for the classification of these tokens. These rules are made by the programmer for his/her convenience. The counting rules also depend upon the programming language.

(c)Function Count.

The size of a large software product can be estimated in a better way, through a larger unit called module, than the LOC measure. A module can be defined as segment of code, which may be compiled independently. For large software systems, it is easier to predict the number of modules than the lines of code. For example, let a software product require n modules. It is generally agreed that size of the module should be about 50 - 60 lines of code. Therefore, size estimate of this software product is about $n \times 60$ lines of code. But, this metric requires precise and strict rules for dividing a program into modules. Due to the absence of these rules, this metric may not be so useful.

A module may consist of one or more functions. In a program, a function may be defined as a group of executable statements, which performs a definite task. The number of lines of code for a function should not be very large. It is because human memory is limited and a programmer cannot perform a task efficiently if the information to be manipulated is large.

LOC metric is like measuring a car stereo by the number of resistors, capacitors, and integrated circuits involved in its production. A drawback.

Here a system is measured based on functionality. This overcomes the disadvantage of using different languages. Function point measures functionality from the users' point of view. Function Point Analysis (FPA) functional units:

Two categories: 1. Transaction function types: External Input(EI), External Output(EO), Enquiries(EQ);
2. Data function types: Internal logical files(ILF), External Interface files(EIF).

COST ESTIMATION: COCOMO MODEL

Before a new software project begins it is needed to study the cost and time it will take. These estimates are often made based solely on past experiences. But this will seldom provide accurate estimation because in many cases the projects are different. Different estimation methods may have the following common attributes – scope of project established in advance, estimates are made based on software metrics, the project is broken into smaller pieces which are estimated individually.

To achieve reliable cost and schedule estimates, one may consider the following possible paths.

- (i) Delay estimation until late in project (This path, however, attractive, is not practical. But it should be noted that the longer we wait the more accurate the estimate would be and the less likely the error in the estimates)
- (ii) Develop empirical models for estimation
- (iii) Acquire one or more automated estimation tools
- (iv) Use simple decomposition techniques to generate project cost and schedule estimates.

Models. A model is concerned with the representation of the process to be estimated. It may be static or dynamic. In a static model, a unique variable (say, size) is taken as a key element for calculating all others (say, cost, time). The form of equation used is the same for all calculations. In a dynamic model, all variables are interdependent and there is no basic variable as in the static model.

When a model makes use of a single basic variable to calculate all others it is said to be a *single-variable model*. Models where several variables are needed to describe the software development process, and selected equations combine these variables to give the estimate of time and cost are referred to as *multivariable models*. The variables, single or multiple, that are input to the model to predict the behaviour of a software development are called *predictors*. The choice and handling of these predictors are most crucial activity in estimating methodologies.

COCOMO Model. COCOMO is a hierarchy of software cost estimation models, which include basic, intermediate and detailed sub models.

Basic model. The basic model aims at estimating, in a quick and rough fashion, the small to medium sized software projects. Three modes of software development are considered in this model – organic, semi-detached and embedded.

In the organic mode, a small team of experienced developers develops software in a very familiar environment. The size of the software development in this mode ranges from small (a few KLOC) to medium (a few tens of KLOC), while in other two models the size ranges from small to very large (a few hundreds of KLOC).

In the embedded mode of software development, the project has tight constraints, which might be related to the target processor and its interface with the associated hardware. The problem to be solved is unique and so it is often hard to find experienced persons, as the same does not usually exist.

The semi-detached mode is an intermediate mode between the organic mode and embedded mode. The comparison of all three modes is given below.

Mode	Project size	Nature of project	Innovation	Deadline of the project	Development environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc	Little	Not tight	Familiar and in house
Semi detached	Typically 50-300 KLOC	Medium size project, medium size team, average previous experience on similar projects. For example, utility systems like compilers, database systems, editors etc	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, real time systems, complex interfaces, very little previous experience. For example, ATMs, Air Traffic Control etc.	Significant	Tight	Complex hardware/customer interfaces required

Depending on the problem at hand, the team might include a mixture of experienced and less experienced people with only a recent history of working together. The basic COCOMO equations take the form

$$E = a_b (\text{KLOC})^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in person-months, and D is the development time in months. The coefficients a_b , b_b , c_b and d_b are given in the table below.

Project	a_b	b_b	c_c	d_d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

When effort and development time are known, the average staff size to complete the project may be calculated as

Average staff size (SS) = (E / D) persons.

When project size is known, the productivity level may be calculated as

Productivity (P) = (KLOC / E) KLOC/PM

With the basic model, the software estimator has a useful tool for estimating quickly, by two runs on a pocket calculator, the cost and development time of a software project, once the size is estimated. The software estimator will have to assess by himself/herself which mode is the most appropriate.

Risk management

Risk. A problem that could cause some loss or threaten the success of the project, but which has not happened yet.

Risk management. The process of identifying, addressing and eliminating these problems before they can damage the project.

Top five risk factors (as pointed out by Capers Jones). Long list of bad things that can happen to a software project. However, Capers Jones pointed out top five risk factors that threatens projects in different applications: Dependencies, Requirement issues, Management issues, Lack of knowledge, Other risk categories.

(a)Dependencies. Many risks arise due to dependencies of project on outside agencies or factors. It is not easy to control these external dependencies. Some typical dependency-related risk factors are: Availability of trained, experienced people; Intercomponent or inter-group dependencies; Customer-furnished items or information; Internal and external subcontractor relationships

(b)Requirements issues. Many projects face uncertainty and turmoil around the product's requirements. While some of this uncertainty is tolerable in early stages, but the threat to success increases if such issues are not resolved as the project progresses. If we do not control requirements-related risk factors, we might either build the wrong product, or build the right product badly. Either situation results in unpleasant surprises and unhappy customers. Some typical factors are: Lack of clear product vision, lack of agreement on product requirements, unprioritized requirements, new markets with uncertain needs, rapidly changing requirements, inadequate impact analysis of requirement changes.

(c)Management Issues. Project managers usually write the risk management plan, and most people do not wish to air their weaknesses (assuming they even recognise them) in public. Nonetheless, issues like those listed below can make it harder for projects to succeed. If we do not confront such touchy issues, we should not be surprised if they bite us at some point. Defined project tracking processes, and clear roles and responsibilities, can address some of these risk factors. Inadequate planning and task identification, inadequate visibility into actual project status, unclear project ownership and decision making, unrealistic commitments made (sometimes for the wrong reasons), managers or customers with unrealistic expectations, staff personality conflicts, poor communication.

(v) Lack of knowledge. The rapid rate of change of technologies, and the increasing change of skilled staff, mean that our project teams may not have the skills we need to be

- successful. The key is to recognise the risk areas early enough so that we can take appropriate preventive actions, such as obtaining training, hiring consultants, and bringing the right people together on the project team. Some of the factors are: inadequate training; poor understanding of methods, tools, and techniques; inadequate application domain experience; new technologies; ineffective, poorly documented, or neglected processes.
- (vi) **Other risk categories.** The list of potential risk areas is long. Some of the critical areas are: unavailability of adequate testing facilities, turnover of essential personnel, unachievable performance requirements, technical approaches that may not work.

Risk management activities:

1. Risk Assessment (Risk identification, Risk analysis, Risk prioritization),
2. Risk control (Risk management planning, Risk monitoring, Risk resolution).

Risk assessment: The process of examining a project and identifying areas of potential risk.

Risk control: The process of managing risks to achieve the desired outcomes.

Unit-V

Software Reliability and Quality Assurance

(i) What is reliability?

- (a) Definition (IEEE): Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time.
- (b) Definition: It is the probability of a failure free operation of a program for a specified time in a specified environment.
- (c) Informal Definition: Software reliability is defined as how the system fulfils the user requirements.

(ii) Software Reliability Vs Hardware Reliability:

(Draw the bath tub curve of hardware reliability: Figure here.)

There are three phases in the life of any hardware component, that is, burn-in, useful life and wear out. In burn in phase, failure rate is quite high initially, and it starts decreasing gradually. During useful life period, failure rate is approximately constant. Failure rate increases in wear-out phase due to wearing out/aging of components.

Software reliability: (Draw the software reliability curve here.)

Software reliability is a different concept as compared to predicting hardware reliability. Unlike hardware, software becomes more reliable over time, instead of wearing out. There is no wear out phase in software. Software may be retired only if it becomes obsolete. Some contributing factors are: change of environment, change of infrastructure/technology, major changes in requirements, increase in complexity, extremely difficult to maintain, deterioration in structure of the code, slow execution speed, poor graphical user interfaces.

Software reliability is not measured on the basis of time, because it does not wear out. There is no problem of rust as like in case of hardware. Electronic and mechanical parts may become old and wear out with time and usage. In the hardware reliability 'time' is used to define the reliability of hardware. It means how much time the hardware remains working without any defect.

Distinct Characteristics of Software and Hardware

Fault- Software faults are mainly design faults where as hardware faults are mostly physical.

Wear out-It is an important point, software remains reliable overtime instead of wearing out like hardware. It becomes obsolete (out of fashion) if the environment for which it is developed changes. Hence software may be retired due to environmental changes, new requirements, new expectations etc.

Software is not manufactured- A software is developed it is not manufactured like hardware. It depends upon the individual skills and creative abilities of the developer which is very difficult to specify and even more difficult to quantify and virtually impossible to standardize.

Time dependency and life cycle- Software reliability is not a function of operational time. But it is applicable on hardware reliability.

Reusability of components- In software graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus and wide variety of interaction mechanism.

But in software where GUI is not used, it used various types of devices. Good software has following features - It should be fault free. It should possess tolerance power. It should be user friendly. It should be within the budget of the customer.

Environmental Factors Environment factors do not affect software reliability, but it affects to the hardware.

(iii) Failures and Faults:

What is software failure? It is the departure of the external results of program operation from requirements. So, failure is dynamic. It relates to the behaviour of the program. It can include things such as deficiency in performance attributes and excessive response time. (A system is said to have a failure if the service it delivers to the user does not fulfil the specification for a specified period of time.)

Fault: A fault is the defect in the program that, when executed under particular conditions, causes a failure. There can be different sets of conditions that cause failures, or the conditions can be repeated. Thus, a fault can be the source of more than one failure. A fault is a property of the program rather than a property of its execution or behaviour. In general, a fault is what we are referring to when we use the term 'bug'. A fault is created when a programmer makes an error.

State the distinctions between software failure and fault?

(iv) **Time:** Reliability quantities are defined with respect to time, although it would be possible to define them with respect to other variables. We are concerned with three kinds of time.

- (a) Execution time: The time that is actually spent by a processor in executing the instructions of that program.
- (b) Calendar time: The familiar time that we normally experience.
- (c) Clock time: It represents the elapsed time from start to end of program execution on a running computer. It includes wait time and the execution time of other programs.

It is generally accepted that models based on execution time are superior. However, quantities must ultimately be related back to calendar time to be meaningful to engineers or managers.

There are four general ways of characterising failure occurrences in time:

- (a) Time of failure
- (b) Time interval between failures
- (c) Cumulative failures experienced up to a given time
- (d) Failures experienced in a time interval.

Time variation can be considered from two different viewpoints, the mean value function and the failure intensity function.

The mean value function represents the average cumulative failures associated with each time point.

The failure intensity function is the rate of change of the mean value function or the number of failures per unit time. For example, you might say 0.01 failure/hr or 1 failure/100hr.

Precisely, the failure intensity is the derivative of the mean value function with respect to time, and is an instantaneous value.

(Draw the mean value and failure intensity function here.)

Failure intensity is an alternative way of expressing reliability.

(Draw the reliability and failure intensity figure here.)

Failure behaviour is affected by two principal factors: (a) the number of faults in the software being executed and (b) the execution environment or the operational profile of execution.

The number of faults in software is the difference between the number introduced and the number removed.

Faults are introduced when the code is being developed by programmers. They may introduce the faults during original design or when they are adding new features, making design changes, or repairing faults that have been identified. In general, only code that is new or modified results in faults introduction. Code that is inherited from another application does not usually introduce any appreciable number of faults, it generally has been thoroughly debugged in the previous application.

Fault removal:

Faults removal clearly can't occur unless you have some means of detecting the fault in the first place. Thus fault removal resulting from execution depends on the occurrence of the associated failure. Occurrence depends both on the length of time for which the software has been executing and on the execution environment or operational profile. When different functions are executed, different faults are encountered and the failures that are exhibited tend to be different; thus environmental influence. We can often find faults without execution. They may be found through inspection, compiler diagnostics, design or code reviews, or code reading.

(v) Uses of software reliability measures:

- (a) An important use of software reliability measurement is in system engineering.
- (b) Software reliability measures can be used to evaluate software engineering technology quantitatively.
- (c) Software reliability measures offer you the possibility of evaluating development status during the test phases of a project.
- (d) You can use software reliability measures to monitor the operational performance of software and to control new features added and design changes made to the software.
- (e) A quantitative understanding of software quality and the various factors influencing it and affected by it enriches the software product and the software development process. One is then much more capable of making informed decisions.

(vi) Reliability Metrics: Reliability metrics are used to quantitatively express the reliability of the software product. The choice of which metric is to be used depends upon the type of system to which it applies & the requirements of the application domain. Measuring the software reliability is a difficult problem because we don't have a good understanding about the nature of software. It is difficult to find a suitable way to measure software reliability, and most of the aspects related to software reliability. Even the software sizes have no uniform definition. If we cannot measure the reliability directly, something can be measured that reflects the characteristics related to reliability.

Some reliability metrics which can be used to quantify the reliability of the software product are discussed below:

(a) Mean Time To Failure (MTTF): MTTF is defined as the time interval between the successive failures. An MTTF of 200 means that one failure can be expected every 200 time units. The time units are totally dependent on the system & it can even be specified in the number of transactions. MTTF is relevant for systems with long transactions. For example, it is suitable for computeraided design systems where a designer will work on a design for several hours as well as for Word-processor systems.

(b) Mean Time To Repair (MTTR): Once the failure occur sometime is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure & to fix them.

(c) Mean Time Between Failure (MTBF): We can combine MTTF & MTTR metrics to get the MTBF metric. $MTBF = MTTF + MTTR$. Thus, an MTBF of 300 indicates that once the failure occurs, the next failure is expected to occur only after 300 hours. In this case the time measurements are real time & not the execution time as in MTTF.

(d) Rate Of Occurrence Of Failure (ROCOF): It is the number of failures occurring in unit time interval. The number of unexpected events over a particular time of operation. ROCOF is the frequency of occurrence with which unexpected behaviour is likely to occur. An ROCOF of 0.02 means that two failures are likely to occur in each 100 operational time unit steps. It is also called failure intensity metric.

(e) Probability Of Failure On Demand (POFOD): POFOD is defined as the probability that the system will fail when a service is requested. It is the number of system failures given a number of systems inputs. POFOD is the likelihood that the system will fail when a service request is made. A POFOD of 0.1 means that one out of a ten service requests may result in failure. POFOD is an important measure for safety critical systems. POFOD is appropriate for protection systems where services are demanded occasionally.

(f) Availability (AVAIL): Availability is the probability that the system is available for use at a given time. It takes into account the repair time & the restart time for the system. An availability of 0.995 means that in every 1000 time units, the system is likely to be available for 995 of these. The percentage of time that a system is available for use, taking into account planned and unplanned downtime. If a system is down an average of four hours out of 100 hours of operation, its AVAIL is 96%. Thus software availability is the probability that a program is operating according to requirement at a given point in time and is defined as

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR} \times 100\%.$$

(vii) Reliability Models:

(a) Basic Execution Time Model:

This model was established by J. D. Musa in 1979, and it is based on execution time. The basic execution model is the most popular and generally used reliability growth model, mainly because of the following reasons:

It is practical, simple, and easy to understand.

Its parameters clearly relate to the physical world.

It can be used for accurate reliability prediction.

The basic execution model determines failure behaviour initially using execution time. Execution time may later be converted into calendar time.

It is equivalent to the Musa-Okumoto (M-O) logarithmic Poisson execution time model, with different mean value function.

The mean value function, in this case, is based on an exponential distribution.

Variables involved in the Basic Execution Model:

Failure intensity (λ): number of failures per time unit.

Execution time (τ): time since the program is running.

Mean failures experienced (μ): mean failures experienced in a time interval.

In the basic execution model, the mean failures experienced μ is expressed in terms of the execution time (τ) as

$$\mu(\tau) = \nu_0 x \left(1 - e^{-\frac{\lambda_0}{\nu_0} \tau} \right)$$

where

λ_0 = stands for the initial failure intensity at the start of the execution

ν_0 = stands for the total number of failures occurring over an infinite time period; it corresponds to the expected number of failures to be observed eventually.

The failure intensity expressed as a function of the execution time is given by

$$\lambda(\tau) = \lambda_0 x e^{-\frac{\lambda_0}{\nu_0} \tau}$$

It is based on the above formula. The failure intensity λ is expressed in terms of μ as:

$$\lambda(\tau) = \lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0} \right)$$

where

λ_0 = Initial

ν_0 = Number of failures experienced, if a program is executed for an infinite time period.

μ = Average or expected number of failures experienced at a given period of time.

τ = Execution time.

(Look up the pdf copy)

(b) Logarithmic Poisson Execution Time Model:

Theta=the rate of reduction in the (normalized) failure intensity per failure.

(c) The Jelinski-Moranda Model:

The Jelinski-Moranda (JM) model, which is also a Markov process model, has strongly affected many later models which are in fact modifications of this simple model.

Characteristics of JM Model:

It is a Binomial type model

It is certainly the earliest and certainly one of the most well-known black-box models.

JM model always yields an over-optimistic reliability prediction.

JM Model follows a perfect debugging step, i.e., the detected fault is removed with certainty simple model.

The constant software failure rate of the JM model at the i^{th} failure interval is given by:

$$\lambda(t_i) = \phi[N - (i - 1)], \quad i = 1, 2, \dots, N \quad (1)$$

where

ϕ = a constant of proportionality indicating the failure rate provided by each fault

N = the initial number of errors in the software

t_i = the time between $(i - 1)^{\text{th}}$ and i^{th} failures.

The mean value and the failure intensity methods for this model which belongs to the binominal type can be obtained by multiplying the inherent number of faults by the cumulative failure and probability density functions (pdf) respectively:

$$\mu(t_i) = N(1 - e^{-\phi t_i}), \quad i = 1, 2, \dots, N \quad (2)$$

$$c(t_i) = N\phi e^{-\phi t_i}, \quad i = 1, 2, \dots, N \quad (3)$$

Those characteristics plus four other characteristics of the J M model are summarized in table:

Measures of Reliability name	Measures of Reliability formula
Probability density function	$f(t_i) = \phi[N - (i - 1)]e^{-\phi[N - (i - 1)]t_i}$
Software reliability function	$R(t_i) = e^{-\phi[N - (i - 1)]t_i}$
Failure rate function	$\lambda(t_i) = \phi[N - (i - 1)]$
Mean time to failure function	$MTTF(t_i) = \frac{1}{\phi[N - (i - 1)]}$
Mean value function	$\mu(t_i) = N(1 - e^{-\phi t_i})$
Failure intensity function	$c(t_i) = N\phi e^{-\phi t_i}$
Median	$m = \{\phi[N - (i - 1)]\}^{-1} \ln 2$
Cumulative distribution function	$f(t_i) = 1 - e^{-\phi[N - (i - 1)]t_i}$

Assumptions

The assumptions made in the J-M model contains the following:

The number of initial software errors is unknown but fixed and constant.
Each error in the software is independent and equally likely to cause a failure during a test.
Time intervals between occurrences of failure are separate, exponentially distributed random variables.
The software failure rate remains fixed over the ranges among fault occurrences.
The failure rate is corresponding to the number of faults that remain in the software.
A detected error is removed immediately, and no new mistakes are introduced during the removal of the detected defect.
Whenever a failure appears, the corresponding fault is reduced with certainty.

Variations in JM Model

JM model was the first prominent software reliability model. Several researchers showed interest and modify this model, using different parameters such as failure rate, perfect debugging, imperfect debugging, number of failures, etc. now, we will discuss different existing variations of this model.

Lipow Modified Version of Jelinski-Moranda Geometric Model

Sukert Modified Schick-Wolverton Model

Schick Wolverton Model

GO Imperfect Debugging Model

Jelinski-Moranda Geometric Model

Little-Verrell Bayesian Model

Shanthikumar General Markov Model

Error Detection Model

Langberg Singpurwalla Model

Jewell Bayesian Software Reliability Model

Quantum Modification to the JM Model
Optimal Software Released Based on Markovian Model
Modification to the JM Model Based on Cloud Model Theory
Modified JM Model with Imperfect Debugging Phenomenon

(d)The Bug Seeding Model:

Bug seeding or debugging is a technique that was developed to evaluate the number of software issues resident in the software product. There are two criteria that explain test and software quality: Bug Detection Rate and Number of Remaining Bugs.

Bug detection rate measures the effectiveness of software testing. Bug detection rate shows the number of faults found by the software tests. Number of remaining bugs is a software metric that reveals how many issues remain after performing the tests.

(viii) Software Quality:

Recall that the objective of software engineering is to produce good quality maintainable software in time and within budget. Different people understand different meanings of quality like: conformance to requirements, fitness for the purpose, level of satisfaction.

The following table give the attribute domain and attributes of software quality:

Attribute Domain	Attributes
Reliability	Correctness
	Consistency and precision
	Robustness
	Simplicity
	Traceability
Usability	Accuracy
	Clarity and accuracy of documentation
	Conformity of operational environment
	Completeness
	Efficiency
Maintainability	Testability
	Accuracy and clarity of documentation
	Modularity
	Readability
Adaptability	Simplicity
	Modifiability
	Expandability
	Portability

The following table is the attributes of software quality:

Reliability	The extent to which a software performs its intended functions without failure.
Correctness	The extent to which a software meets its specifications.
Consistency and precision	The extent to which a software is consistent and give results with precision.
Robustness	The extent to which a software tolerates the unexpected problems.
Simplicity	The extent to which a software is simple in its operations.
Traceability	The extent to which an error is traceable in order to fix it.
Usability	The extent of effort required to learn, operate and

	understand the functions of the software
Accuracy	Meeting specifications with precision.
Clarity and accuracy of documentation	The extent to which documents are clearly & accurately written.
Conformity of operational environment	The extent to which a software is in conformity of operational environment.
Completeness	The extent to which a software has specified functions.
Efficiency	The amount of computing resources and code required by software to perform a function.
Testability	The effort required to test a software to ensure that it performs its intended functions.
Maintainability	The effort required to locate and fix an error during maintenance phase.
Adaptability	The extent to which a software is adaptable to new platforms & technologies.
Modularity	It is the extent of ease to implement, test, debug and maintain the software.
Readability	The extent to which a software is readable in order to understand.
Modifiability	The effort required to modify a software during maintenance phase.
Expandability	The extent to which a software is expandable without undesirable side effects.
Portability	The effort required to transfer a program from one platform to another platform.

(ix) Software Quality Models: Quality models consist of a number of quality characteristics (or factors as called in some models). These quality characteristics could be used to reflect the quality of the software product from the view of that characteristic.

What does it actually do?

To promote the quality of the software under development. A pre-inspection to have a better product.

Given the intangible and abstract nature of software, researchers and practitioners have been looking for ways to characterize software in order to make the benefits and costs more visible (for measurement). This quest continues today but there have been two notable models of software quality attributes, namely:- McCall's Quality Model(1977) and Boehm SQM (1978).

There are other models (see FURPS) but these two illustrate the general purpose and issues of these quality factor models. These two quality models are summarized below and similarities can be seen.

To begin with there are some common objectives of these models, namely:-

The benefits and costs of software are represented in their totality with no overlap between the attributes.

The presence, or absence, of these attributes can be measured objectively.

The degree to which each of these attributes is present reflects the overall quality of the software product.

These attribute facilitate continuous improvement, allowing cause and effect analysis which maps to these attributes, or measure of the attribute.

(a) McCall Software Quality Model

Jim McCall produced this model for the US Air Force and the aim was to bridge the gap between users and developers. He tried to map the user view with the developer's priority.

McCall identified three main perspectives for characterizing the quality attributes of a software product.

These perspectives are:-

Product revision (ability to change).

Product transition (adaptability to new environments).

Product operations (basic operational characteristics).

Product Revision: The product revision perspective identifies quality factors that influence the ability to change the software product, these factors are:

Maintainability, the ability to find and fix a defect.

Flexibility, the ability to make changes required as dictated by the business.

Testability, the ability to Validate the software requirements.

Product Transition: The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments:

Portability, the ability to transfer the software from one environment to another.

Reusability, the ease of using existing software components in a different context.

Interoperability, the extent, or ease, to which software components work together.

Product Operations: The product operations perspective identifies quality factors that influence the extent to which the software fulfils its specification:

Correctness, the functionality matches the specification.

Reliability, the extent to which the system fails.

Efficiency, system resource (including cpu, disk, memory, network) usage.

Integrity, protection from unauthorized access.

Usability, ease of use.

In total McCall identified 11 quality factors (attributes) broken down by the three perspectives, as listed above.

For each quality factor McCall defined one or more quality criteria (a way of measurement), in this way an overall quality assessment could be made of a given software product by evaluating the criteria for each factor.

For example, the *maintainability* quality factor would have the criteria of *simplicity*, *conciseness* and *modularity*.

The main objective of the McCall's Quality Model is that the quality factors structure should provide a complete software quality

McCall's Quality Model: Major Perspectives 1, 2, 3

Major Perspective 1: Quality Factors 1, 2, 3

Major Perspective 2:

.....

Quality Factor 1: Quality Criteria 1, 2, 3

.....

.....

Quality Criteria 1: Metrics 1, 2, 3

Major Perspective 1: Product Revision (it is about the ability of the product to undergo changes)

Major Perspective 2: Product Operations (It is about the characteristics of the product operation)

Major Perspective 3: Product Transition (Adaptability of the product to new environments)

(b) Boehm Software Quality Model

Barry W. Boehm also defined a hierarchical model of software quality characteristics, in trying to qualitatively define software quality as a set of attributes and metrics (measurements). At the highest level of his model, Boehm defined three primary uses (or basic software requirements), these three primary uses are

As-is utility, the extent to which the as-is software can be used (i.e. ease of use, reliability and efficiency).

Maintainability, ease of identifying what needs to be changed as well as ease of modification and retesting.

Portability, ease of changing software to accommodate a new environment.

These three *primary uses* had quality factors associated with them, representing the next level of Boehm's hierarchical model.

Boehm identified seven quality factors, namely:-

- Portability, the extent to which the software will work under different computer configurations (i.e. operating systems, databases etc.).
- Reliability, the extent to which the software performs as required, i.e. the absence of defects.
- Efficiency, optimum use of system resources during correct execution.

- Usability, ease of use.
- Testability, ease of validation, that the software meets the requirements.
- Understandability, the extent to which the software is easily comprehended with regard to purpose and structure.
- Flexibility, the ease of changing the software to meet revised requirements.

[These quality factors are further broken down into *Primitive constructs* that can be measured, for example *Testability* is broken down into *accessibility*, *communicativeness*, *structure* and *self-descriptiveness*. As with McCall's Quality Model, the intention is to be able to measure the lowest level of the model. (For detail lists of primitive constructs, see figure given in slides).]

Summary of the above two models

Although only a summary of the two example software factor models has been given, some comparisons and observations can be made that generalize the overall quest to characterize software.

Both of McCall and Boehm models follow a similar structure, with a similar purpose. They both attempt to breakdown the software artifact into constructs that can be measured. Some quality factors are repeated, for example: usability, portability, efficiency and reliability.

The presence of more or less factors is not, however, indicative of a better or worse model.

The value of these, and other models, is purely a pragmatic one and not in the semantics or structural differences.

The extent to which one model allows for an accurate measurement (cost and benefit) of the software will determine its value.

The ISO 9126 model of software characteristics represents the ISO's recent attempt to define a set of useful quality characteristics.

(c) **ISO 9126:** ISO 9126 is an international standard for the evaluation of software. The standard is divided into four parts which addresses, respectively, the following subjects: *quality model*; *external metrics*; *internal metrics*; and *quality in use metrics*. ISO 9126 Part one, referred to as ISO 9126-1 is an extension of previous work done by McCall (1977), Boehm (1978), FURPS and others in defining a set of software quality characteristics.

ISO 9126-1 represents the latest (and ongoing) research into characterizing software for the purposes of software quality control, software quality assurance and software process improvement (SPI). We shall discuss the characteristics identified by ISO 9126-1. The other parts of ISO 9126, concerning metrics or measurements for these characteristics, are essential for SQC, SQA and SPI but the main concern here is the definition of the basic ISO 9126 Quality Model.

The ISO 9126 documentation itself, from the official ISO 9126 documentation, can only be purchased and is subject to copyright. SQA.net only reproduces the basic structure of the ISO 9126 standard and any descriptions, commentary or guidance are original material based on public domain information as well as our own experience. The ISO 9126-1 software quality model identifies 6 main quality characteristics, namely:

Functionality, Reliability, Usability, Efficiency, Maintainability, Portability

These characteristics are broken down into sub-characteristics, a high level table is shown below. It is at the sub-characteristic level that measurement for SPI will occur. The main characteristics of the ISO9126-1 quality model, can be defined as follow:-

Functionality

Functionality is the essential purpose of any product or service. For certain items this is relatively easy to define, for example a ship's anchor has the function of holding a ship at a given location. The more functions a product has, e.g. an ATM machine, then the more complicated it becomes to define it's functionality. For software a list of functions can be specified, i.e. a sales order processing systems should be able to record customer information so that it can be used to reference a sales order. A sales order system should also provide the following functions: Record sales order product, price and quantity.

Calculate total price.

Calculate appropriate sales tax.

Calculate date available to ship, based on inventory.

Generate purchase orders when stock falls below a given threshold.

The list goes on and on but the main point to note is that functionality is expressed as a totality of essential functions that the software product provides. It is also important to note that the presence or absence of these functions in a software product can be verified as either existing or not, in that it is a Boolean (either a yes or no answer). The other software characteristics listed (i.e. usability) are only present to some degree, i.e. not a simple on or off. Many people get confused between overall process functionality (in which software plays a part) and software functionality. This is partly due to the fact that Data Flow Diagrams (DFDs) and other modeling tools can depict process functionality (as a set of data in\data out conversions) and software functionality. Consider a sales order process, that has both manual and software components. A function of the sales order process could be to record the sales order but we could implement a hard copy filing cabinet for the actual orders and only use software for calculating the price, tax and ship date. In this way the functionality of the software is limited to those calculation functions. SPI, or Software Process Improvement is different from overall Process Improvement or Process Re-engineering, ISO 9126-1 and other software quality models do not help measure overall Process costs\benefits but only the software component. The relationship between software functionality within an overall business process is outside the scope of ISO 9126 and it is only the software functionality, or essential purpose of the software component, that is of interest for ISO 9126.

Following functionality, there are 5 other software attributes that characterize the usefulness of the software in a given environment.

Each of the following characteristics can only be measured (and are assumed to exist) when the functionality of a given system is present. In this way, for example, a system can not possess usability characteristics if the system does not function correctly (the two just don't go together).

Reliability

Once a software system is functioning, as specified, and delivered the reliability characteristic defines the capability of the system to maintain its service provision under defined conditions for defined periods of time. One aspect of this characteristic is fault tolerance that is the ability of a system to withstand component failure. For example, if the network goes down for 20 seconds then comes back the system should be able to recover and continue functioning.

Usability

Usability only exists with regard to functionality and refers to the ease of use for a given function. For example, a function of an ATM machine is to dispense cash as requested. Placing common amounts on the screen for selection, i.e. \$20.00, \$40.00, \$100.00 etc, does not impact the function of the ATM but addresses the Usability of the function. The ability to learn how to use a system (learnability) is also a major sub-characteristic of usability.

Efficiency

This characteristic is concerned with the system resources used when providing the required functionality. The amount of disk space, memory, network etc. provides a good indication of this characteristic. As with a number of these characteristics, there are overlaps. For example, the usability of a system is influenced by the system's Performance, in that if a system takes 3 hours to respond the system would not be easy to use although the essential issue is a performance or efficiency characteristic.

Maintainability

The ability to identify and fix a fault within a software component is what the maintainability characteristic addresses. In other software quality models this characteristic is referenced as supportability. Maintainability is impacted by code readability or complexity as well as modularization. Anything that helps with identifying the cause of a fault and then fixing the fault is the concern of maintainability. Also the ability to verify (or test) a system, i.e. testability, is one of the sub-characteristics of maintainability.

Portability

This characteristic refers to how well the software can adopt to changes in its environment or with its requirements. The sub-characteristics of this characteristic include adaptability. Object oriented design and implementation practices can contribute to the extent to which this characteristic is present in a given system.

The full table of characteristics and sub-characteristics for the ISO 9126-1 Quality Model is:-

Characteristics	Sub-characteristics	Definitions
Functionality	Suitability	This is the essential Functionality characteristic and refers to the appropriateness (to specification) of the functions of the software.
	Accurateness	This refers to the correctness of the functions, an ATM may provide a cash dispensing function but is the amount correct?
	Interoperability	A given software component or system does not typically function in isolation. This subcharacteristic concerns the ability of a software component to interact with other components or systems.

	Compliance	Where appropriate certain industry (or government) laws and guidelines need to be complied with, i.e. SOX. This subcharacteristic addresses the compliant capability of software.
	Security	This sub-characteristic relates to unauthorized access to the software functions.
Reliability	Maturity	This subcharacteristic concerns frequency of failure of the software.
	Fault tolerance	The ability of software to withstand (and recover) from component, or environmental, failure.
	Recoverability	Ability to bring back a failed system to full operation, including data and network connections.
Usability	Understandability	Determines the ease of which the systems functions can be understood, relates to user mental models in Human Computer Interaction methods.
	Learnability	Learning effort for different users, i.e. novice, expert, casual etc.
	Operability	Ability of the software to be easily operated by a given user in a given environment.
Efficiency	Time behaviour	Characterizes response times for a given thru put, i.e. transaction rate.
	Resource behaviour	Characterizes resources used, i.e. memory, cpu, disk and network usage.
Maintainability	Analyzability	Characterizes the ability to identify the root cause of a failure within the software.
	Stability	Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.
	Testability	Characterizes the effort needed to verify (test) a system change.
	Changeability	Characterizes the amount of effort to change a system.
Portability	Adaptability	Characterizes the ability of the system to change to new specifications or operating environments.
	Installability	Characterizes the effort required to install the software.
	Conformance	Similar to compliance for functionality, but this characteristic relates to portability. One example would be Open SQL conformance which relates to portability of database used.
	Replaceability	Characterizes the plug and play aspect of software components, that is how easy is it to exchange a given software component within a specified environment.

ISO 9126 Observations:

For the most part, the overall structure of ISO9126-1 is similar to past models, McCall (1977) and Boehm (1978), although there are a couple of notable differences. Compliance comes under the functionality characteristic, this can be attributed to government initiatives like SOX. In many requirements specifications all characteristics, that are specified, that are not pure functional requirements are specified as Non-Functional requirements. It is interesting to note, with ISO9126, that compliance is seen as a functional characteristic.

Using the ISO 9126 (or any other quality model) for derivation of system requirements brings clarity of definition of purpose and operating capability.

A designer typically will need to make trade-offs between two or more characteristics when designing the system. Consider highly modularized code, this code is usually easy to maintain, i.e. has a good changeability characteristic, but may not perform as well (for cpu resource, as unstructured program code).

Although ISO 9126-1 is the latest proposal for a useful Quality Model, of software characteristics, it is unlikely to be the last. One thing is certain, the requirements (including compliance) and operating environment of software will be continually changing and with this change will come the continuing search to find useful characteristics that facilitate measurement and control of the software production process.

(x) **ISO 9000 Certification for Software Industry:**

ISO (International Standards Organization) is a group of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

Types of ISO 9000 Quality Standards: ISO 9000 is a series of three standards, namely, ISO 9001, ISO 9002 and ISO 9003.

The ISO 9000 series of standards is based on the assumption that if a proper stage is followed for production, then good quality products are bound to follow automatically. The types of industries to which the various ISO standards apply are as follows.

ISO 9001: This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.

ISO 9002: This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products.

Therefore, ISO 9002 does not apply to software development organizations.

ISO 9003: This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

How to get ISO 9000 certification?

An organization wanting to obtain ISO 9000 certification applies to ISO registrar office for registration. The process consists of the following stages:

1. **Applications:** Once an organization decided to go for ISO certification, it applies to the registrar for registration.
2. **Pre-Assessment:** During this stage, the registrar makes a rough assessment of the organization.
3. **Document review and Adequacy of Audit:** During this stage, the registrar reviews the document submitted by the organization and suggest an improvement.
4. **Compliance Audit:** During this stage, the registrar checks whether the organization has compiled the suggestion made by it during the review or not.
5. **Registration:** The Registrar awards the ISO certification after the successful completion of all the phases.
6. **Continued Inspection:** The registrar continued to monitor the organization time by time.:

(xi) **SEI Capability Maturity Model:** The capability maturity model (CMM) is not a software life cycle model. Instead, it is a strategy for improving the software process, irrespective of the actual life cycle model used. The CMM was developed by Software Engineering Institute (SEI) of Carnegie-Mellon University in 1986.

CMM is used to judge the maturity of the software processes of an organisation and to identify the key practices that are required to increase the maturity of these processes.

The Capability Maturity Model (CMM) is a procedure used to develop and refine an organization's software development process.

The model defines a five-level evolutionary stage of increasingly organized and consistently more mature processes.

Capability Maturity Model is used as a benchmark to measure the maturity of an organization's software process.

(The CMM is organized into five maturity levels as shown in the following figure.)

(Draw the maturity levels of CMM here)

There are two methods of SEICMM:

Capability Evaluation and Software Process Assessment

Capability Evaluation: Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicate the likely contractor performance if the contractor is awarded a work. Therefore, the results of the software process capability assessment can be used to select a contractor.

Software Process Assessment: Software process assessment is used by an organization to improve its process capability. Thus, this type of evaluation is for purely internal use.

SEI CMM categorized software development industries into the following five maturity levels. The various levels of SEI CMM have been designed so that it is easy for an organization to build its quality system starting from scratch slowly.

Maturity Levels:

Level 1: Initial (Processes unpredictable, poorly controlled and reactive)

Ad hoc activities characterize a software development organization at this level. Very few or no processes are described and followed. Since software production processes are not limited, different engineers follow their process and as a result, development efforts become chaotic. Therefore, it is also called a chaotic level.

Level 2: Repeatable (Processes characterized for projects and is often reactive)

At this level, the fundamental project management practices like tracking cost and schedule are established. Size and cost estimation methods, like function point analysis, COCOMO, etc. are used.

Level 3: Defined (Processes characterized for the organisation and is proactive. Projects tailors their processes from organisation's standards)

At this level, the methods for both management and development activities are defined and documented. There is a common organization-wide understanding of operations, roles, and responsibilities. The ways through defined, the process and product qualities are not measured. ISO 9000 goals at achieving this level.

Level 4: Managed (Processes measured and controlled)

At this level, the focus is on software metrics. Two kinds of metrics are composed.

Product metrics measure the features of the product being developed, such as its size, reliability, time complexity, understandability, etc.

Process metrics follow the effectiveness of the process being used, such as average defect correction time, productivity, the average number of defects found per hour inspection, the average number of failures detected during testing per LOC, etc. The software process and product quality are measured, and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to analyze if a project performed satisfactorily. Thus, the outcome of process measurements is used to calculate project performance rather than improve the process.

Level 5: Optimizing (Focus on process improvement)

At this phase, process and product metrics are collected. Process and product measurement data are evaluated for continuous process improvement.

Key Process Areas (KPA) of a software organization

Except for SEI CMM level 1, each maturity level is featured by several Key Process Areas (KPAs) that contains the areas an organization should focus on improving its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig.

CMM Level	Focus	Key Process Areas
1. Initial	Competent people	No KPAs

2. Repeatable	Project Management	Software project planning, Software configuration management
3. Defined	Definition of processes	Process definition, Training program, Peer reviews
4. Managed	Product and process quality	Quantitative process metrics, Software quality management
5. Optimizing	Continuous process improvement	Defect prevention, Process change management, Technology change management

SEI CMM provides a series of key areas on which to focus to take an organization from one level of maturity to the next. Thus, it provides a method for gradual quality improvement over various stages. Each step has been carefully designed such that one step enhances the capability already built up.

People Capability Maturity Model (PCMM): Outside syllabus

UNIT VI SOFTWARE TESTING AND MAINTENANCE

What is testing?

Some definitions usually given by people:

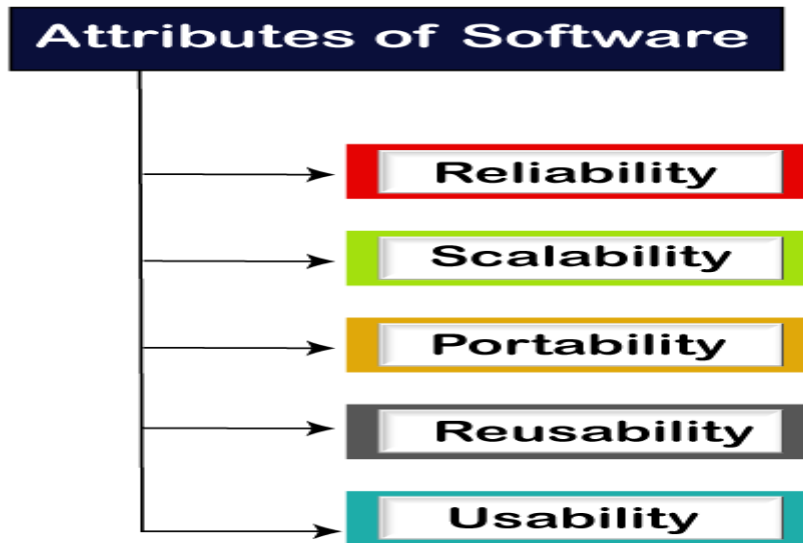
- (i) Testing is the process of establishing confidence that a program does what it is supposed to do.
- (ii) Testing is the process of demonstrating that errors are not present.
- (iii) The purpose of testing is to show that a program performs its intended functions correctly.

However, these definitions are not the correct.

The correct definition of testing:

Testing is the process of executing a program with the intent of finding errors.

Software testing is a process of identifying the correctness of software by considering all its attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.



Why testing?

Testing is itself an expensive activity. But launching a software without testing may lead to cost potentially much higher than that of testing, especially in systems where human safety is involved.

In software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

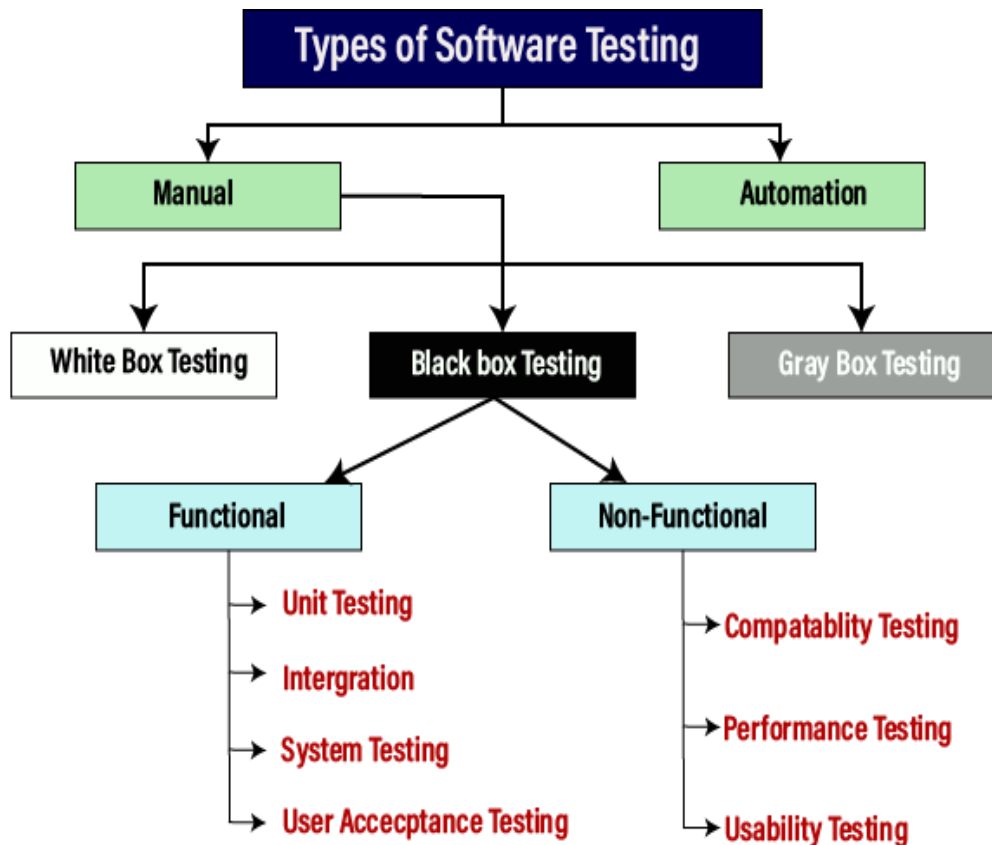
Who should conduct the testing?

Testing requires the developers to find errors from their software. It is hard for software developers to point out errors from own creations. Thus many organisations made a distinction between development and testing phase by making different people responsible for each phase.

What should be tested?

One should test the program's responses to every possible input. That is, one should test for all valid and invalid inputs.

Types of testing:



Manual testing

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

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

1. **White box testing/Structural Testing:** The box testing approach of software testing consists of black box testing and white box testing. White box testing is also known as *glass box testing*, *structural testing*, *clear box testing*, *open box testing* and *transparent box testing*.

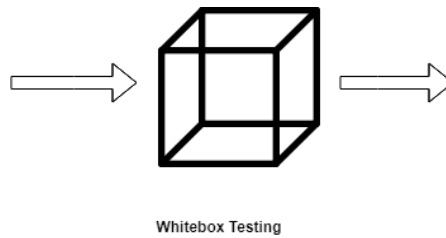
It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs.

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 term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Test cases for white box testing are derived from the design phase of the software development lifecycle. Data flow testing, control flow testing, path testing, branch testing, statement and decision coverage all these techniques used by white box testing as a guideline to create an error-free software.



White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Generic steps of white box testing

Design all test scenarios, test cases and prioritize them according to high priority number.

This step involves the study of code at runtime to examine the resource utilization, not accessed areas of the code, time taken by various methods and operations and so on.

In this step testing of internal subroutines takes place. Internal subroutines such as nonpublic methods, interfaces are able to handle all types of data appropriately or not.

This step focuses on testing of control statements like loops and conditional statements to check the efficiency and accuracy for different data inputs.

In the last step white box testing includes security testing to check all possible security loopholes by looking at how the code handles security.

Reasons for white box testing

It identifies internal security holes.

To check the way of input inside the code.

Check the functionality of conditional loops.

To test function, object, and statement at an individual level.

Advantages of White box testing

White box testing optimizes code so hidden errors can be identified.

Test cases of white box testing can be easily automated.

This testing is more thorough than other testing approaches as it covers all code paths.

It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

White box testing is too much time consuming when it comes to large-scale programming applications.

White box testing is much expensive and complex.

It can lead to production error because it is not detailed by the developers.

White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation.

Techniques Used in White Box Testing

Data Flow Testing	Data flow testing is a group of testing strategies that examines the control flow of programs in order to explore the sequence of variables according to the sequence of events.
Control Flow Testing	Control flow testing determines the execution order of statements or instructions of the program through a control structure. The control structure of a program is used to develop a test case for the program. In this technique, a particular part of a large program is selected by the

	tester to set the testing path. Test cases represented by the control graph of the program.
Branch Testing	Branch coverage technique is used to cover all branches of the control flow graph. It covers all the possible outcomes (true and false) of each condition of decision point at least once.
Statement Testing	Statement coverage technique is used to design white box test cases. This technique involves execution of all statements of the source code at least once. It is used to calculate the total number of executed statements in the source code, out of total statements present in the source code.
Decision Testing	This technique reports true and false outcomes of Boolean expressions. Whenever there is a possibility of two or more outcomes from the statements like do while statement, if statement and case statement (Control flow statements), it is considered as decision point because there are two outcomes either true or false.

2. Black box testing:

Black box testing is a technique of software testing **which examines the functionality of software** without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer.

In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.



Generic steps of black box testing

The black box test is based on the specification of requirements, so it is examined in the beginning.

In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.

In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.

The fourth phase includes the execution of all test cases.

In the fifth step, the tester compares the expected output against the actual output.

In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Test procedure

The test procedure of black box testing is a kind of process in which the tester has specific knowledge about the software's work, and it develops test cases to check the accuracy of the software's functionality.

It does not require programming knowledge of the software. All test cases are designed by considering the input and output of a particular function. A tester knows about the definite output of a particular input, but not about how the result is arising. There are various techniques used in black box testing for testing like decision table technique, boundary value analysis technique, state transition, All-pair testing, cause-effect graph technique, equivalence partitioning technique, error guessing technique, use case technique and user story technique. All these techniques have been explained in detail within the tutorial.

Test cases

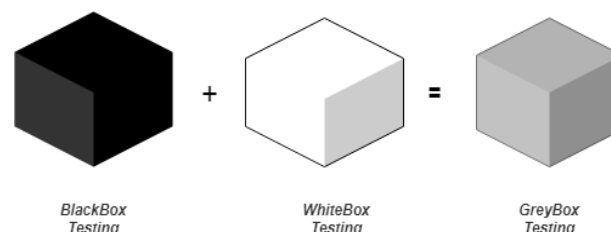
Test cases are created considering the specification of the requirements. These test cases are generally created from working descriptions of the software including requirements, design parameters, and other specifications. For the testing, the test designer selects both positive test scenario by taking valid input values and adverse test scenario by taking invalid input values to determine the correct output. Test cases are mainly designed for functional testing but can also be used for non-functional testing. Test cases are designed by the testing team, there is not any involvement of the development team of software.

Techniques Used in Black Box Testing

Decision Table Technique	Decision Table Technique is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form. It is appropriate for the functions that have a logical relationship between two and more than two inputs.
Boundary Value Technique	Boundary Value Technique is used to test boundary values, boundary values are those that contain the upper and lower limit of a variable. It tests, while entering boundary value whether the software is producing correct output or not.
State Transition Technique	State Transition Technique is used to capture the behavior of the software application when different input values are given to the same function. This applies to those types of applications that provide the specific number of attempts to access the application.
All-pair Testing Technique	All-pair testing Technique is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input, list box, text box, etc.
Cause-Effect Technique	Cause-Effect Technique underlines the relationship between a given result and all the factors affecting the result. It is based on a collection of requirements.
Equivalence Partitioning Technique	Equivalence partitioning is a technique of software testing in which input data divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behaviour.
Error Guessing Technique	Error guessing is a technique in which there is no specific method for identifying the error. It is based on the experience of the test analyst, where the tester uses the experience to guess the problematic areas of the software.
Use Case Technique	Use case Technique used to identify the test cases from the beginning to the end of the system as per the usage of the system. By using this technique, the test team creates a test scenario that can exercise the entire software based on the functionality of each function from start to end.

3. Grey box testing:

Greybox testing is a software testing method to test the software application with partial knowledge of the internal working structure. It is a combination of black box and white box testing because it involves access to internal coding to design test cases as white box testing and testing practices are done at functionality level as black box testing.



GreyBox testing commonly identifies context-specific errors that belong to web systems. For example; while testing, if tester encounters any defect then he makes changes in code to resolve the defect and then test it again

in real time. It concentrates on all the layers of any complex software system to increase testing coverage. It gives the ability to test both presentation layer as well as internal coding structure. It is primarily used in integration testing and penetration testing.

Why Grey Box testing?

Reasons for Grey Box testing are as follows.

It provides combined benefits of both Black Box testing and White Box testing.

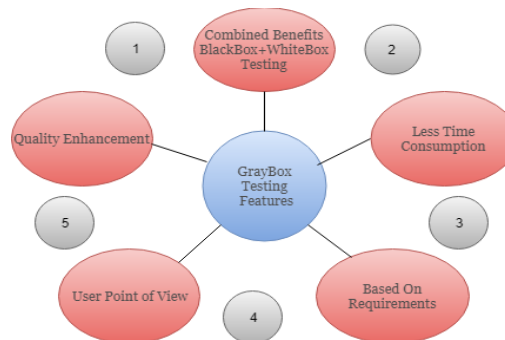
It includes the input values of both developers and testers at the same time to improve the overall quality of the product.

It reduces time consumption of long process of functional and non-functional testing.

It gives sufficient time to the developer to fix the product defects.

It includes user point of view rather than designer or tester point of view.

It involves examination of requirements and determination of specifications by user point of view deeply.



Grey Box Testing Strategy

Grey box testing does not make necessary that the tester must design test cases from source code. To perform this testing test cases can be designed on the base of, knowledge of architectures, algorithm, internal states or other high -level descriptions of the program behavior. It uses all the straightforward techniques of black box testing for function testing. The test case generation is based on requirements and preset all the conditions before testing the program by assertion method.

Generic Steps to perform Grey box testing are:

First, select and identify inputs from Black Box and White Box testing inputs.

Second, I identify expected outputs from these selected inputs.

Third, identify all the major paths to traverse through during the testing period.

The fourth task is to identify sub-functions which are the part of main functions to perform deep level testing.

The fifth task is to identify inputs for sub-functions.

The sixth task is to identify expected outputs for sub-functions.

The seventh task includes executing a test case for Sub-functions.

The eighth task includes verification of the correctness of result.

The test cases designed for Greybox testing includes Security related, Browser related, GUI related, Operational system related and Database related testing.

Techniques of Grey box Testing

Matrix Testing

This testing technique comes under Grey Box testing. It defines all the used variables of a particular program. In any program, variable are the elements through which values can travel inside the program. It should be as per requirement otherwise, it will reduce the readability of the program and speed of the software. Matrix technique is a method to remove unused and uninitialized variables by identifying used variables from the program.

Regression Testing

Regression testing is used to verify that modification in any part of software has not caused any adverse or unintended side effect in any other part of the software. During confirmation testing, any defect got fixed, and that part of software started working as intended, but there might be a possibility that fixed defect may have introduced a different defect somewhere else in the software. So, regression testing takes care of these type of defects by testing strategies like retest risky use cases, retest within a firewall, retest all, etc.

Orthogonal Array Testing or OAT

The purpose of this testing is to cover maximum code with minimum test cases. Test cases are designed in a way that can cover maximum code as well as GUI functions with a smaller number of test cases.

Pattern Testing

Pattern testing is applicable to such type of software that is developed by following the same pattern of previous software. In these type of software possibility to occur the same type of defects. Pattern testing determines reasons of the failure so they can be fixed in the next software.

Usually, automated software testing tools are used in Grey box methodology to conduct the test process. Stubs and module drivers provided to a tester to relieve from manually code generation.

Automation testing

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

Verification and Validation

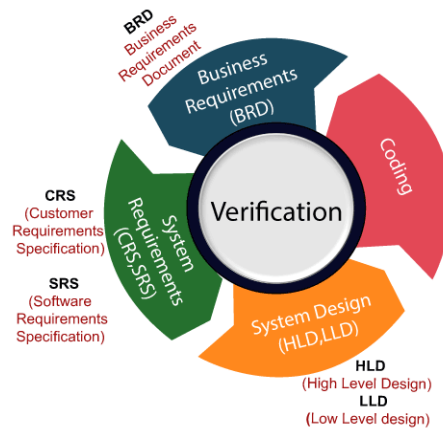
Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements.

Testing = Verification + Validation

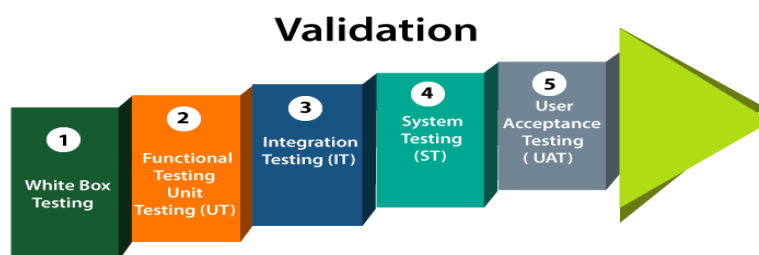
Verification testing includes different activities such as business requirements, system requirements, design review, and code walkthrough while developing a product.

It is also known as static testing, where we are ensuring that "we are developing the right product or not". And it also checks that the developed application fulfilling all the requirements given by the client.



Validation testing is testing where tester performed functional and non-functional testing. Here functional testing includes Unit Testing (UT), Integration Testing (IT) and System Testing (ST), and non-functional testing includes User Acceptance Testing (UAT).

Validation testing is also known as dynamic testing, where we are ensuring that "we have developed the product right." And it also checks that the software meets the business needs of the client.



Difference between verification and validation testing

Verification	Validation
We check whether we are developing the right product or not.	We check whether the developed product is right.
Verification is also known as static testing.	Validation is also known as dynamic testing.
Verification includes different methods like Inspections, Reviews, and Walkthroughs.	Validation includes testing like functional testing, system testing, integration, and User acceptance testing.
It is a process of checking the work-products (not the final product) of a development cycle to decide whether the product meets the specified requirements.	It is a process of checking the software during or at the end of the development cycle to decide whether the software follow the specified business requirements.
Quality assurance comes under verification testing.	Quality control comes under validation testing.
The execution of code does not happen in the verification testing.	In validation testing, the execution of code happens.
In verification testing, we can find the bugs early in the development phase of the product.	In the validation testing, we can find those bugs, which are not caught in the verification process.
Verification testing is executed by the Quality assurance team to make sure that the product is developed according to customers' requirements.	Validation testing is executed by the testing team to test the application.
Verification is done before the validation testing.	After verification testing, validation testing takes place.

In this type of testing, we can verify that the inputs follow the outputs or not.	In this type of testing, we can validate that the user accepts the product or not.
-----------------------------------------------------------------------------------	------------------------------------------------------------------------------------

Code Inspection

Code Inspection is the most formal type of review, which is a kind of static testing to avoid the defect multiplication at a later stage.

The main purpose of code inspection is to find defects and it can also spot any process improvement if any.

An inspection report lists the findings, which include metrics that can be used to aid improvements to the process as well as correcting defects in the document under review.

Preparation before the meeting is essential, which includes reading of any source documents to ensure consistency.

Inspections are often led by a trained moderator, who is not the author of the code.

The inspection process is the most formal type of review based on rules and checklists and makes use of entry and exit criteria.

It usually involves peer examination of the code and each one has a defined set of roles.

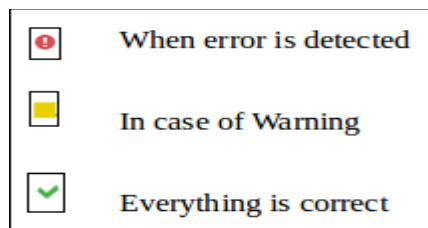
Inspection is a static code analysis tool which helps us to find runtime errors, locate dead code, detect performance issue, memory leak, spelling problems and also improve the overall code structure. It does not only tell us about where is the problem in the code but also suggest the corrections right away. Code Inspection are available when we perform code analysis for -

The Whole Project

Particular File

Custom Scope

By default, IntelliJ IDEA performs analysis on all open files. If an error is detected in the project then we will see the following icon in the top right side of the Editor.



Running Code Inspections

There are two ways to run Code Inspection:

1. Inspect Code

Open the file in the Editor.

Go to Analyse -> Inspect Code.

Dialog Box Open. Select one of the following -

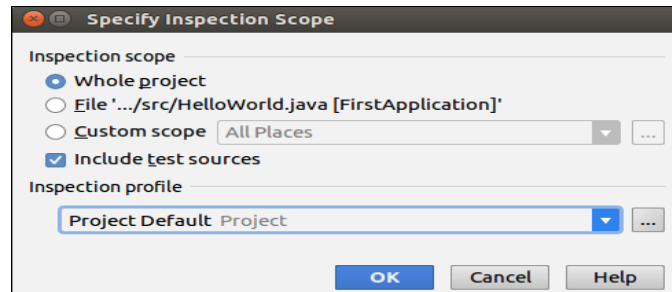
The Whole Projects

Particular File

Custom Scope

Specify the Inspection Profile

Click **Ok** button.



2. Run Inspection by Name

Open the file in the Editor.

Go to Analyse -> Run Inspection By Name.

Dialog Box open. Enter the Inspection Name

New Window will open. Select one of the following -

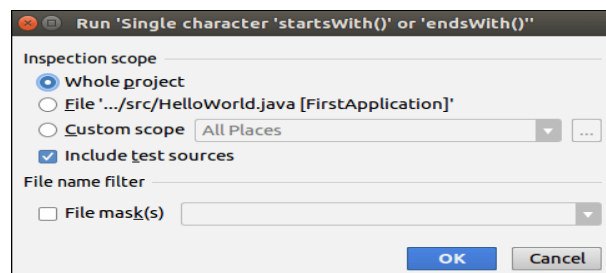
The Whole Projects

Particular File

Custom Scope

Select the **Include test sources** checkbox.

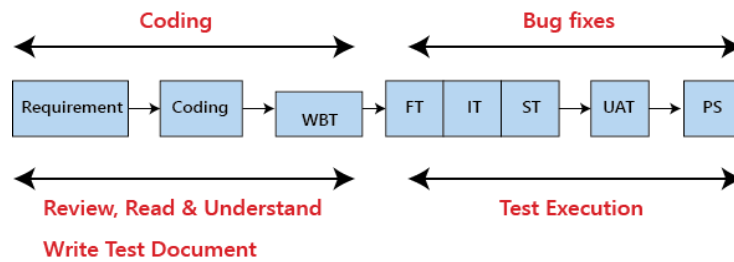
Select the **File Masks**, if we want to apply Inspection only in that files which have matching the specific Masks. Click **Ok**.



Testing Documentation

Testing documentation is the documentation of artefacts that are created during or before the testing of a software application. Documentation reflects the importance of processes for the customer, individual and organization.

Projects which contain all documents have a high level of maturity. Careful documentation can save the time, efforts and wealth of the organization.



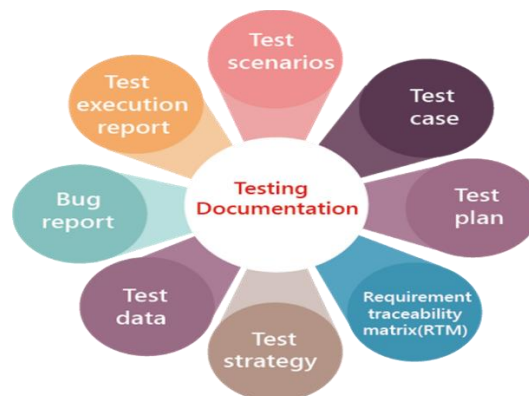
There is the necessary reference document, which is prepared by every test engineer before stating the test execution process. Generally, we write the test document whenever the developers are busy in writing the code.

Once the test document is ready, the entire test execution process depends on the test document. The primary objective for writing a test document is to decrease or eliminate the doubts related to the testing activities.

Types of test document

In software testing, we have various types of test document, which are as follows:

- Test scenarios
- Test case
- Test plan
- Requirement traceability matrix(RTM)
- Test strategy
- Test data
- Bug report
- Test execution report



Test Scenarios

It is a document that defines the multiple ways or combinations of testing the application. Generally, it is prepared to understand the flow of an application. It does not consist of any inputs and navigation steps.

Test case

It is an in-details document that describes step by step procedure to test an application. It consists of the complete navigation steps and inputs and all the scenarios that need to be tested for the application. We write the test case to maintain the consistency, or every tester will follow the same approach for organizing the test document.

Test plan

It is a document that is prepared by the managers or test lead. It consists of all information about the testing activities. The test plan consists of multiple components such as Objectives, Scope, Approach, Test Environments, Test methodology, Template, Role & Responsibility, Effort estimation, Entry and Exit criteria, Schedule, Tools, Defect tracking, Test Deliverable, Assumption, Risk, and Mitigation Plan or Contingency Plan.

Requirement Traceability Matrix (RTM)

The Requirement traceability matrix [RTM] is a document which ensures that all the test case has been covered. This document is created before the test execution process to verify that we did not miss writing any test case for the particular requirement.

Test strategy

The test strategy is a high-level document, which is used to verify the test types (levels) to be executed for the product and also describe that what kind of technique has to be used and which module is going to be tested. The Project Manager can approve it. It includes the multiple components such as documentation formats, objective, test processes, scope, and customer communication strategy, etc. we cannot modify the test strategy.

Test data

It is data that occurs before the test is executed. It is mainly used when we are implementing the test case. Mostly, we will have the test data in the Excel sheet format and entered manually while performing the test case.

The test data can be used to check the expected result, which means that when the test data is entered, the expected outcome will meet the actual result and also check the application performance by entering the incorrect input data.

Bug report

The bug report is a document where we maintain a summary of all the bugs which occurred during the testing process. This is a crucial document for both the developers and test engineers because, with the help of bug reports, they can easily track the defects, report the bug, change the status of bugs which are fixed successfully, and also avoid their repetition in further process.

Test execution report

It is the document prepared by test leads after the entire testing execution process is completed. The test summary report defines the constancy of the product, and it contains information like the modules, the number of written test cases, executed, pass, fail, and their percentage. And each module has a separate spreadsheet of their respective module.

Why documentation is needed

If the testing or development team gets software that is not working correctly and developed by someone else, so to find the error, the team will first need a document. Now, if the documents are available then the team will quickly find out the cause of the error by examining documentation. But, if the documents are not available then the tester need to do black box and white box testing again, which will waste the time and money of the organization. More than that, lack of documentation becomes a problem for acceptance.

Example

Let's take a real-time example of Microsoft, Microsoft launch every product with proper user guidelines and documents, which are very explanatory, logically consistent and easy to understand for any user. These are all the reasons behind their successful products.

Benefits of using Documentation

Documentation clarifies the quality of methods and objectives.

It ensures internal coordination when a customer uses software application.

It ensures clarity about the stability of tasks and performance.

It provides feedback on preventive tasks.

It provides feedback for your planning cycle.

It creates objective evidence for the performance of the quality management system.

If we write the test document, we can't forget the values which we put in the first phase.

It is also a time-saving process because we can easily refer to the text document.

It is also consistent because we will test on the same value.

The drawback of the test document

It is a bit tedious because we have to maintain the modification provided by the customer and parallel change in the document.

If the test documentation is not proper, it will replicate the quality of the application.

Sometimes it is written by that person who does not have the product knowledge.

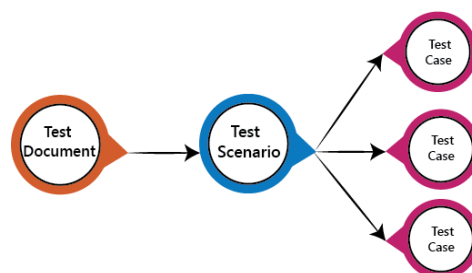
Sometimes the cost of the document will be exceeding its value.

Test Scenarios

The test scenario is a detailed document of test cases that cover end to end functionality of a software application in liner statements. The liner statement is considered as a scenario. The test scenario is a high-level classification of testable requirements. These requirements are grouped on the basis of the functionality of a module and obtained from the use cases.

In the test scenario, there is a detailed testing process due to many associated test cases. Before performing the test scenario, the tester has to consider the test cases for each scenario.

In the test scenario, testers need to put themselves in the place of the user because they test the software application under the user's point of view. Preparation of scenarios is the most critical part, and it is necessary to seek advice or help from customers, stakeholders or developers to prepare the scenario.



How to write Test Scenarios

As a tester, follow the following steps to create Test Scenarios-

Read the requirement document such as BRS (Business Requirement Specification), SRS (System Requirement Specification) and FRS (Functional Requirement Specification) of the software which is under the test.

Determine all technical aspects and objectives for each requirement.

Find all the possible ways by which the user can operate the software.

Ascertain all the possible scenario due to which system can be misused and also detect the users who can be hackers.

After reading the requirement document and completion of the scheduled analysis make a list of various test scenarios to verify each function of the software.

Once you listed all the possible test scenarios, create a traceability matrix to find out whether each and every requirement has a corresponding test scenario or not.

Supervisor of the project reviews all scenarios. Later, they are evaluated by other stakeholders of the project.

Features of Test Scenario

The test scenario is a liner statement that guides testers for the testing sequence.

Test scenario reduces the complexity and repetition of the product.

Test scenario means talking and thinking about tests in detail but write them in liner statements.

It is a thread of operations.

Test scenario becomes more important when the tester does not have enough time to write test cases, and team members agree with a detailed liner scenario.

The test scenario is a time saver activity.

It provides easy maintenance because the addition and modification of test scenarios are easy and independent.

Some rules have to be followed when we were writing test scenarios:

Always list down the most commonly used feature and module by the users.

We always start the scenarios by picking module by module so that a proper sequence is followed as well as we don't miss out on any module level.

Generally, scenarios are module level.

Delete scenarios should always be the last option else, and we will waste lots of time in creating the data once again.

It should be written in a simple language.

Every scenario should be written in one line or a maximum of two lines and not in the paragraphs.

Every scenario should consist of Dos and checks.

Example of Test scenarios

Here we are taking the Gmail application and writing test scenarios for different modules which are most commonly used such as Login, Compose, Inbox, and Trash.

Test scenarios on the Login module

Enter the valid login details (Username, password), and check that the home page is displayed.

Enter the invalid Username and password and check for the home page.

Leave Username and password blank, and check for the error message displayed.

Enter the valid Login, and click on the cancel, and check for the fields reset.

Enter invalid Login, more than three times, and check that account blocked.

Enter valid Login, and check that the Username is displayed on the home screen.

Test scenarios on Compose module

Checks that all users can enter email ids in the To, Cc, and Bcc.

Check that the entire user can enter various email ids in To, Cc, and Bcc.

Compose a mail, send it, and check for the confirmation message.

Compose a mail, send it, and check in the sent item of the sender and the inbox.

Compose a mail, send it, and check for invalid and valid email id (valid format), check the mail in sender inbox.

Compose mail, discard, and then check for confirmation message and check-in draft.

Compose mail click on save as draft and check for the confirmation message

Compose mail click on close and check for confirmation save as drafts.

Test scenarios on Inbox module

Click on the inbox, and verify all received mail are displayed and highlighted in the inbox.

Check that a latest received mail has been displayed to the sender email id correctly.

Select the mail, reply and forward send it; check in the sent item of sender and inbox of the receiver.

Check for any attached attachments to the mail that are downloaded or not.

Check that attachment is scanned correctly for any viruses before download.

Select the mail, reply and forward save as draft, and check for the confirmation message and checks in the Draft section.

Check all the emails are marked as read are not highlighted.

Check all mail recipients in Cc are visible to all users.

Checks all email recipients in Bcc are not visible to the users.

Select mail, delete it, and then check in the Trash section.

Test scenario on Trash module

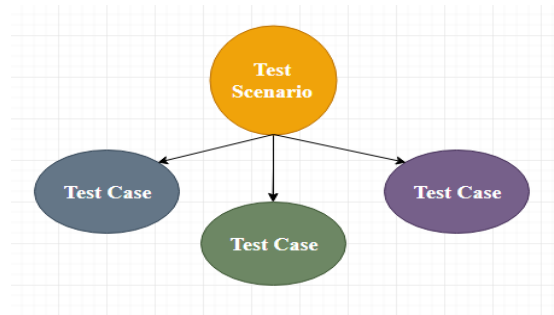
Open trash, check all deleted mail present.

Restore mail from Trash; check-in the corresponding module.

Select mail from trash, delete it, and check mail is permanently deleted.

Test Case

The test case is defined as a group of conditions under which a tester determines whether a software application is working as per the customer's requirements or not. Test case designing includes preconditions, case name, input conditions, and expected result. A test case is a first level action and derived from test scenarios.



It is an in-details document that contains all possible inputs (positive as well as negative) and the navigation steps, which are used for the test execution process. Writing of test cases is a one-time attempt that can be used in the future at the time of regression testing.

Test case gives detailed information about testing strategy, testing process, preconditions, and expected output. These are executed during the testing process to check whether the software application is performing the task for that it was developed or not.

Test case helps the tester in defect reporting by linking defect with test case ID. Detailed test case documentation works as a full proof guard for the testing team because if developer missed something, then it can be caught during execution of these full-proof test cases.

To write the test case, we must have the requirements to derive the inputs, and the test scenarios must be written so that we do not miss out on any features for testing. Then we should have the test case template to maintain the uniformity, or every test engineer follows the same approach to prepare the test document.

Generally, we will write the test case whenever the developer is busy in writing the code.

When do we write a test case?

We will write the test case when we get the following:

When the customer gives the business needs then, the developer starts developing and says that they need 3.5 months to build this product.

And in the meantime, the testing team will start writing the test cases.

Once it is done, it will send it to the Test Lead for the review process.

And when the developers finish developing the product, it is handed over to the testing team.

The test engineers never look at the requirement while testing the product document because testing is constant and does not depends on the mood of the person rather than the quality of the test engineer.

Note: When writing the test case, the actual result should never be written as the product is still being in the development process. That's why the actual result should be written only after the execution of the test cases.

Why write the test cases?

We will write the test for the following reasons:

To require consistency in the test case execution

To make sure a better test coverage

It depends on the process rather than on a person

To avoid training for every new test engineer on the product

To require consistency in the test case execution: we will see the test case and start testing the application.

To make sure a better test coverage: For this, we should cover all possible scenarios and document it, so that we need not remember all the scenarios again and again.

It depends on the process rather than on a person: A test engineer has tested an application during the first release, second release, and left the company at the time of third release. As the test engineer understood a module and tested the application thoroughly by deriving many values. If the person is not there for the third release, it becomes difficult for the new person. Hence all the derived values are documented so that it can be used in the future.

To avoid giving training for every new test engineer on the product: When the test engineer leaves, he/she leaves with a lot of knowledge and scenarios. Those scenarios should be documented so that the new test engineer can test with the given scenarios and also can write the new scenarios.

Note: When the developers are developing the first product for the First release, the test engineer writes the test cases. And in the second release, when the new features are added, the test engineer writes the test cases for that also, and in the next release, when the elements are modified, the test engineer will change the test cases or writes the new test cases as well.

Test Case Template

The primary purpose of writing a test case is to achieve the efficiency of the application.

Header

Test Case Name/ID :-Release - Version - Application Name - Module

Test Case Type:-

E.T.C	I.T.C	S.T.C
-------	-------	-------

Requirement Number:-

Module:-

Severity:- Critical/Major/Minor

Status:-

Release:-

Version:-

Pre-condition:-

Test Data:-

Summary:-

Body

Step No.	Description	Inputs	Expected Result	Actual Result	Status	Comments
...
...

Footer

Author:- Reviewd By:-

Date:- Approved By:-

As we know, the actual result is written after the test case execution, and most of the time, it would be same as the expected result. But if the test step will fail, it will be different. So, the actual result field can be skipped, and in the Comments section, we can write about the bugs.

And also, the Input field can be removed, and this information can be added to the Description field.

The above template we discuss above is not the standard one because it can be different for each company and also with each application, which is based on the test engineer and the test lead. But, for testing one application, all the test engineers should follow a usual template, which is formulated.

The test case should be written in simple language so that a new test engineer can also understand and execute the same.

In the above sample template, the header contains the following:

Step number. It is also essential because if step number 20 is failing, we can document the bug report and hence prioritize working and also decide if it's a critical bug.

Test case type. It can be functional, integration or system test cases or positive or negative or positive and negative test cases.

Release. One release can contain many versions of the release.

Pre-condition. These are the necessary conditions that need to be satisfied by every test engineer before starting the test execution process. Or it is the data configuration or the data setup that needs to be created for the testing.

For example: In an application, we are writing test cases to add users, edit users, and delete users. The per-condition will be seen if user A is added before editing it and removing it.

Test data. These are the values or the input we need to create as per the per-condition.

For example, Username, Password, and account number of the users.

The test lead may be given the test data like username or password to test the application, or the test engineer may themselves generate the username and password.

Severity. The severity can be major, minor, and critical, the severity in the test case talks about the importance of that particular test cases. All the test execution process always depends on the severity of the test cases. We can choose the severity based on the module. There are many features include in a module, even if one element is critical, we claim that test case to be critical. It depends on the functions for which we are writing the test case.

For example, we will take the Gmail application and let us see the severity based on the modules:

Modules	Severity
Login	Critical
Help	Minor
Compose mail	Critical
Setting	Minor
Inbox	Critical
Sent items	Major
Logout	Critical

And for the banking application, the severity could be as follows:

Modules	Severity
Amount transfer	Critical
Feedback	Minor

Brief description. The test engineer has written a test case for a particular feature. If he/she comes and reads the test cases for the moment, he/she will not know for what feature has written it. So, the brief description will help them in which feature test case is written.

(Discuss an example of a test case template here)

Types of test cases

We have a different kind of test cases, which are as follows:

Function test cases

Integration test cases

System test cases

The functional test cases

Firstly, we check for which field we will write test cases and then describe accordingly. In functional testing or if the application is data-driven, we require the input column else; it is a bit time-consuming.

Rules to write functional test cases:

In the expected results column, try to use should be or must be.

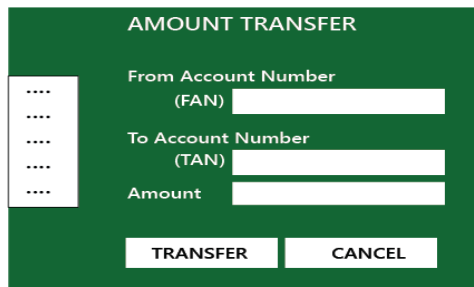
Highlight the Object names.

We have to describe only those steps which we required the most; otherwise, we do not need to define all the steps.

To reduce the excess execution time, we will write steps correctly.

Write a generic test case; do not try to hard code it.

Let say it is the amount transfer module, so we are writing the functional test cases for it and then also specifies that it is not a login feature.



The functional test case for amount transfer module is in the below Excel file:

Functional Test case template						
Test case name	beta-1.0-ICD-amount transfer					
Test case type	Functional test case					
Requirement no	G					
Module	amount transfer					
Status	XXX					
Severity	Critical					
Release	Beta					
Version	1					
Pre-condition	sender login two account number Balance--> exist					
Test data	Username:xyz, Password:1234 1231, 4321 3000-9000					
Summary	to check the functionality of amount transfer					
Steps no	Description	Inputs	Expected result	Actual results	Status	Comments
1	Open "Browser" and enter the "Url"	https://QA/Main/it/	Login page" must be display	As Expected	pass	XXX
2	Enter the following values for "Username" and "Password" and click on the "OK" button	xyz, 1234	"Home page" must be displayed	As Expected	pass	XXX
3	Click on the "Amount Transfer"	Null			pass	XXX
4	Enter the following for From Account number (FAN):					
	valid	1234	Accept	As Expected	pass	
	invalid	1124	Error message invalid account		fail	
	blank		Error message FAN value cannot be blank		fail	
			test maximum coverage			
5	Enter the following values for "TO account number (TAN)					
	valid	4321	Accept	As expected	pass	XXX
	invalid	6655	Error message invalid account		fail	
	Blank		Error message TAN value cannot be blank			
			test maximum coverage			
6	enter the value for "Amount"					
	valid	5000, 5001, 9000, 84	Accept	As expected	pass	XXX
	invalid	4999, 9001	error message amount should be between (5000-9000)		fail	
7	Enter the value for "FAN, TAN, Amount" click on the "Transfer" button					
	FAN	1234	"Confirmation Message" amount transfer successfully must be displayed			XXX
	TAN	4321		As expected	pass	
	Amount	6000				
8	Enter the value for "FAN, TAN, Amount" click on the "Cancel" button					
	FAN	1234				
	TAN	4321	All field must be cleared	As expected	pass	XXX
	Amount	6000				
Author	Sem					
Date	4/1/2020					
Reviewed by	jessica					
Approved by	ryan					

Integration test case

In this, we should not write something which we already covered in the functional test cases, and something we have written in the integration test case should not be written in the system test case again.

Rules to write integration test cases

Firstly, understand the product

Identify the possible scenarios

Write the test case based on the priority

When the test engineer writing the test cases, they may need to consider the following aspects:

If the test cases are in details:

They will try to achieve maximum test coverage.

All test case values or scenarios are correctly described.

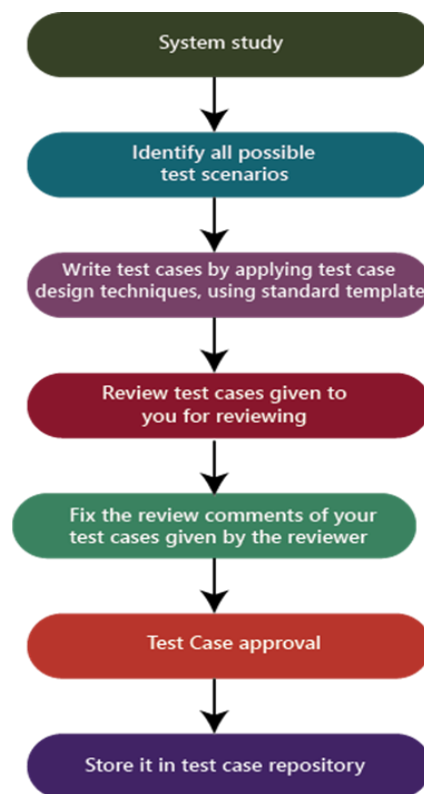
They will try to think about the execution point of view.

The template which is used to write the test case must be unique.

Note: when we involve fewer numbers of steps or coverage is more, it should be the best test case, and when these test cases are given to anyone, they will understand easily.

System test cases. We will write the system test cases for the end-to-end business flows. And we have the entire modules ready to write the system test cases.

The process to write test cases. The method of writing a test case can be completed into the following steps, which are as below:



System study. In this, we will understand the application by looking at the requirements or the SRS, which is given by the customer.

Identify all scenarios:

When the product is launched, what are the possible ways the end-user may use the software to identify all the possible ways.

I have documented all possible scenarios in a document, which is called test design/high-level design.

The test design is a record having all the possible scenarios.

Write test cases. Convert all the identified scenarios to test claims and group the scenarios related to their features, prioritize the module, and write test cases by applying test case design techniques and use the standard test case template, which means that the one which is decided for the project.

Review the test cases. Review the test case by giving it to the head of the team and, after that, fix the review feedback given by the reviewer.

Test case approval. After fixing the test case based on the feedback, send it again for the approval.

Store in the test case repository. After the approval of the particular test case, store in the familiar place that is known as the test case repository.

Test Plan

A test plan is a detailed document which describes software testing areas and activities. It outlines the test strategy, objectives, test schedule, required resources (human resources, software, and hardware), test estimation and test deliverables.

The test plan is a base of every software's testing. It is the most crucial activity which ensures availability of all the lists of planned activities in an appropriate sequence.

The test plan is a template for conducting software testing activities as a defined process that is fully monitored and controlled by the testing manager.

Types of Test Plan

There are three types of the test plan

- Master Test Plan

- Phase Test Plan

- Testing Type Specific Test Plans

Master test plan is a type of test plan that has multiple levels of testing. It includes a complete test strategy.

A **phase test plan** is a type of test plan that addresses any one phase of the testing strategy. For example, a list of tools, a list of test cases, etc.

Specific test plan designed for major types of testing like security testing, load testing, performance testing, etc. In other words, a specific test plan designed for non-functional testing.

How to write a Test Plan

Making a test plan is the most crucial task of the test management process. According to IEEE 829, follow the following seven steps to prepare a test plan.

- First, analyze product structure and architecture.

- Design the test strategy.

- Define all the test objectives.

- Define the testing area.

- Define all the useable resources.

- Schedule all activities in an appropriate manner.

- Determine all the Test Deliverables.

Test plan components or attributes

The test plan consists of various parts, which help us to derive the entire testing activity.



Objectives: It consists of information about modules, features, test data etc., which indicate the aim of the application means the application behavior, goal, etc.

Scope: It contains information that needs to be tested with respective of an application. The Scope can be further divided into two parts: In scope, Out scope.

In scope: These are the modules that need to be tested rigorously (in-detail).

Out scope: These are the modules, which need not be tested rigorously.

Approach: It will explain the flow of the testing an application. It will be classified into two parts

Top to bottom approach

Bottom to top approach

Test Environments: These are the environments where we will test the application, and here we have two types of environments, which are of software and hardware configuration.

The **software configuration** means the details about different Operating Systems such as Windows, Linux, UNIX, and Mac and various Browsers like Google Chrome, Firefox, Opera, Internet Explorer, and so on. And the **hardware configuration** means the information about different sizes of RAM, ROM, and the Processors.

Test methodology: It contains information about performing a different kind of testing like Functional testing, Integration testing, and System testing, etc. on the application.

Template: It is a format used to maintain Generic, and there are different types of the template used during the entire testing process such as

Test case Template

Test case review template

RTM Template

Bug Report Template

Role & Responsibility: It defines the complete task which needs to be performed by the entire testing team.

Let us consider one example where we will understand the roles and responsibility of the Test manager, test lead, and the test engineers.

Role: Test Manager

Name: Ryan

Responsibility:

- Prepare (write and review) the test plan
- Conduct the meeting with the development team
- Conduct the meeting with the testing team
- Conduct the meeting with the customer
- Conduct one monthly stand up meeting
- Handling Escalations

Role: Test Lead

Name: Harvey

Responsibility:

- Prepare (write and review) the test plan
- Conduct daily stand up meeting
- Review and approve the test case
- Prepare the RTM and Reports

Assign modules

Handling schedule

Role: Test Engineer 1, Test Engineer 2 and Test Engineer 3

Name: Louis, Jessica, Donna

Assign modules: M1, M2, and M3

Responsibility:

Write, Review, and Execute the test documents which consists of test case and test scenarios

Read, review, understand and analysis the requirement

Write the flow of the application

Execute the test case

RTM for respective modules

Defect tracking

Effort estimation: In this, we will plan the effort need to be applied by every team member.

Entry and Exit criteria: It is a necessary condition, which needs to be satisfied before starting and stopping the testing process.

Entry Criteria:

The entry criteria contain the following conditions:

Understand and analyze the requirement and prepare the test documents or when the test documents are ready.

Build or the application must be prepared

Modules or features need to be assigned to the different test engineers.

The necessary resource must be ready.

Exit Criteria:

The exit criteria contain the following conditions:

When all the test cases are executed.

Most of the test cases must be passed.

There must not be any blocker or major bug, whereas some minor bugs exist.

Schedule: It is used to explain the timing to work, which needs to be done.

For example:

Writing test cases

Execution process

Tools: There are different types of tools that are used during the testing process, such as Test management tools, Defect tracking tools, Automation testing tools, Unit testing tools, and so on.

Defect tracking: It is generally done with the help of tools because we cannot track the status of each bug manually. Some of the most commonly used bug tracking tools are Jira, Bugzilla, Mantis, and Trac, etc.

Test Deliverable: These are the documents, which we handed over to the customer along with the product. It includes the following:

Test Cases

Test Scripts

RTM

Test Execution Report

Release Notes

Note: *The release note consists of User manual, Installation process, List of the platform on which Tested/list of available bugs in the current release, and the list of fixed bugs in the previous release.*

Assumption: It contains information about a problem or issue which may be occurred during the testing process.

Risk: These are the challenges which we need to face to test the application in the current release.

For example, the effect for an application, release date becomes postponed.

Mitigation Plan or Contingency Plan: It is a back-up plan which is prepared to overcome the risks or issues.

Test Plan Guidelines

Collapse your test plan.

Avoid overlapping and redundancy.

If you think that you do not need a section that is already mentioned above, then delete that section and proceed ahead.

Be specific. For example, when you specify a software system as the part of the test environment, then mention the software version instead of only name.

Avoid lengthy paragraphs.

Use lists and tables wherever possible.

Update plan when needed.

Do not use an outdated and unused document.

Importance of Test Plan

The test plan gives direction to our thinking. This is like a rule book, which must be followed.

The test plan helps in determining the necessary efforts to validate the quality of the software application under the test.

The test plan helps those people to understand the test details that are related to the outside like developers, business managers, customers, etc.

Important aspects like test schedule, test strategy, test scope etc are documented in the test plan so that the management team can review them and reuse them for other similar projects.

Test Case Specification

After successfully achieving software testing objectives, the team starts preparing several reports and documents that are delivered to the client after the culmination of the testing process. During this stage these documents and reports are created by the team, which include details about the testing process, its techniques, methods, test data, test environment, test suite, etc. Test Case Specification document, which is the last document published by the testing team is a vital part of these deliverables and is mainly developed by the organization responsible for formally testing the software product or application. We shall discuss here the importance, format and specification of this document.

What is Test Case Specification in software testing?

One of the deliverables offered to the client, test case specification is a document that delivers a detailed summary of what scenarios will be tested in a software during the software testing life cycle (STLC). This document specifies the main objective of a specific test and identifies the required inputs as well as expected results/outputs.

Format for Test Case Specifications:

1. **Objectives:** This phase comprises of the main purpose of testing software. The relevant and crucial information that makes process easier to understand is mentioned in the objectives.
2. **Pre-conditions:** The items and documents required before executing a particular test case is mentioned here like Requirement Specification, Detail Design Specification, User Guides, Operations Manual and System Design Specifications.
3. **Input Specifications:** Once the pre-conditions are defined, the team works together to identify all the inputs that are required for executing the test cases.
4. **Output Specifications:** These include all outputs that are required to verify the test case such as data, tables, human actions, conditions, files, timings, etc.
5. **Post Conditions:** Here the team defines the various environment, special requirements and constraints on the test cases as stated by the team after the process of testing. It consist of hardware, software, procedural requirements, etc.
6. **Intercase Dependencies:** In this phase the team identifies any requirements or prerequisites test cases. To ensure the quality of these cases the team identifies follow on test cases.

Being one of the deliverables offered to the client after the development of the software, it is vital for test case specification document to be prepared properly with accurate and complete information about the various executed test cases and scenarios.

Developed by the organization that is responsible for formal testing of the software, the test case specification document needs to be prepared separately for each unit to ensure its effectivity and to help build a proper and efficient test plan. Therefore, the format that is used for creating this document is:

Objectives: The purpose of testing the software is defined here in detail. Relevant and crucial information, which can make the process understandable for the reader is mainly included in this section of the format.

Preconditions: The items and documents that were required before executing a particular test case is mentioned with proper evidence and records. Moreover, it describes the features and conditions required for testing. Other important details included here are:

Requirement Specification.

Detail Design Specification.

User Guides.

Operations Manual.

System Design Specifications.

Input Specifications: Once the preconditions are defined, the team works together to identify all the inputs that are required for executing the test cases. These can vary on the basis of the level the test case is written for.

Output Specification: These include all outputs that are required to verify the test case such as data, tables, human actions, conditions, files, timings, etc.

Post Conditions: Here, the team defines the various environment requirements stated by the team after the process of testing. Moreover, the team identifies the any special requirements and constraints on the test cases. It consists of details like:

Hardware: configuration and limitations.

Software: System, operating system, tools, etc.

Procedural Requirements: Special setup, output location & identification, operations interventions, etc.

Others: Mix of applications, facilities, trainings, among other things.

Intercase Dependencies: Finally, the team identifies any requirements or prerequisites test cases. Here, the test cases are documented with references other requirements and specifications. To ensure the quality of these test cases the team identifies follow on test cases.

Conclusion: Being one of the deliverables offered to the client after the development of the software, it is vital for test case specification document to be prepared properly with accurate and complete information about the various executed test cases and scenarios. Moreover, to validate the quality of the whole process, the team provides authorized and approved documents, which further help them improve the effectiveness of the whole report. Hence, with the assistance of this report, software testers can easily define the specifications of testing as well as the process of testing to the client.

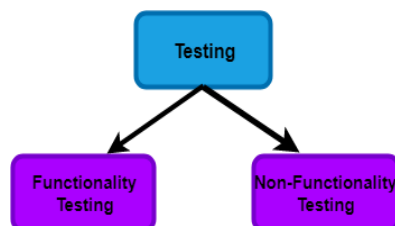
Functional Testing

Recall: Testing is to compare the actual result with the expected result. Testing is done to identify whether all the function is working as expectations. Software testing is a technique to check whether the actual result matches the expected result and to ensure that the software has not any defect or bug.

Software testing ensures that the application has not any defect or the requirement is missing to the actual need. Either manual or automation testing can do software testing.

Software testing also defines as verification of application under test (AUT).

There are two types of (Black box) testing:



Functional testing is a type of software testing which is used to verify the functionality of the software application, whether the function is working according to the requirement specification. In functional testing, each function tested by giving the value, determining the output, and verifying the actual output with the expected value. Functional testing performed as black-box testing which is presented to confirm that the

functionality of an application or system behaves as we are expecting. It is done to verify the functionality of the application.

Functional testing also called as black-box testing, because it focuses on application specification rather than actual code. Tester has to test only the program rather than the system.

Goal of functional testing. The purpose of the functional testing is to check the primary entry function, necessarily usable function, the flow of screen GUI. Functional testing displays the error message so that the user can easily navigate throughout the application.

Process of functional testing. Testers follow the following steps in the functional testing:

Tester does verification of the requirement specification in the software application.

After analysis, the requirement specification tester will make a plan.

After planning the tests, the tester will design the test case.

After designing the test, case tester will make a document of the traceability matrix.

The tester will execute the test case design.

Analysis of the coverage to examine the covered testing area of the application.

Defect management should do to manage defect resolving.



What to test in functional testing?

The main objective of functional testing is checking the functionality of the software system. It concentrates on:

Basic Usability: Functional Testing involves the usability testing of the system. It checks whether a user can navigate freely without any difficulty through screens.

Accessibility: Functional testing test the accessibility of the function.

Mainline function: It focuses on testing the main feature.

Error Condition: Functional testing is used to check the error condition. It checks whether the error message displayed.

Process to perform functional testing:

There are the following steps to perform functional testing:

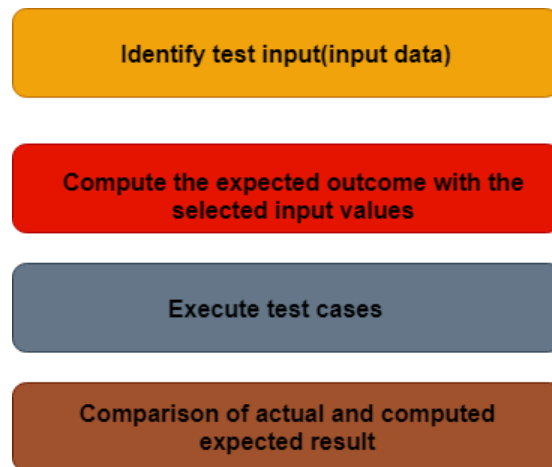
There is a need to understand the software requirement.

Identify test input data

Compute the expected outcome with the selected input values.

Execute test cases

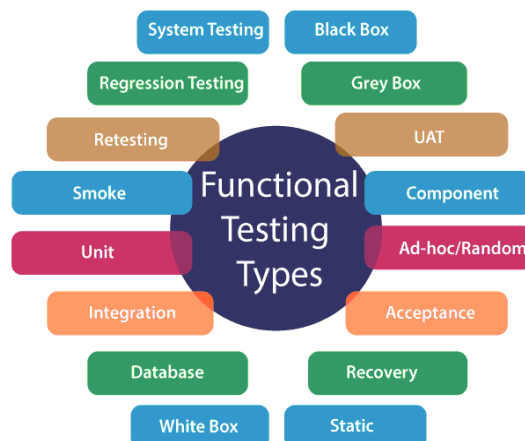
Comparison between the actual and the computed result



Types of functional testing.

The main objective of functional testing is to test the functionality of the component. Functional testing is divided into multiple parts.

Here are the following types of functional testing.



Unit Testing: Unit testing is a type of software testing, where the individual unit or component of the software tested. Unit testing, examine the different part of the application, by unit testing functional testing also done, because unit testing ensures each module is working correctly.

The developer does unit testing. Unit testing is done in the development phase of the application.

Smoke Testing: Functional testing by smoke testing. Smoke testing includes only the basic (feature) functionality of the system. Smoke testing is known as "Build Verification Testing." Smoke testing aims to ensure that the most important function work. For example, Smoke testing verifies that the application launches successfully will check that GUI is responsive.

Sanity Testing: Sanity testing involves the entire high-level business scenario is working correctly. Sanity testing is done to check the functionality/bugs fixed. Sanity testing is little advance than smoke testing. For example, login is working fine; all the buttons are working correctly; after clicking on the button navigation of the page is done or not.

Regression Testing: This type of testing concentrate to make sure that the code changes should not side effect the existing functionality of the system. Regression testing specifies when bug arises in the system after fixing

the bug, regression testing concentrate on that all parts are working or not. Regression testing focuses on is there any impact on the system.

Integration Testing: Integration testing combined individual units and tested as a group. The purpose of this testing is to expose the faults in the interaction between the integrated units. Developers and testers perform integration testing.

White box testing: White box testing is known as Clear Box testing, code-based testing, structural testing, extensive testing, and glass box testing, transparent box testing. It is a software testing method in which the internal structure/design/ implementation tested known to the tester. The white box testing needs the analysis of the internal structure of the component or system.

Black box testing: It is also known as behavioural testing. In this testing, the internal structure/ design/ implementation not known to the tester. This type of testing is functional testing. Why we called this type of testing is black-box testing, in this testing tester, can't see the internal code.

For example, a tester without the knowledge of the internal structures of a website tests the web pages by using the web browser providing input and verifying the output against the expected outcome.

User acceptance testing: It is a type of testing performed by the client to certify the system according to requirement. The final phase of testing is user acceptance testing before releasing the software to the market or production environment. UAT is a kind of black-box testing where two or more end-users will involve.

Retesting: Retesting is a type of testing performed to check the test cases that were unsuccessful in the final execution are successfully pass after the defects fixed. Usually, tester assigns the bug when they find it while testing the product or its component. The bug allocated to a developer, and he fixes it. After fixing, the bug is assigned to a tester for its verification. This testing is known as retesting.

Database Testing: Database testing is a type of testing which checks the schema, tables, triggers, etc. of the database under test. Database testing may involve creating complex queries to load/stress test the database and check its responsiveness. It checks the data integrity and consistency. Example: let us consider a banking application whereby a user makes a transaction. Now from database testing following, things are important. They are:

- Application store the transaction information in the application database and displays them correctly to the user.

- No information lost in this process

- The application does not keep partially performed or aborted operation information.

- The user information is not allowed individuals to access by the

Ad-hoc testing: Ad-hoc testing is an informal testing type whose aim is to break the system. This type of software testing is unplanned activity. It does not follow any test design to create the test cases. Ad-hoc testing is done randomly on any part of the application; it does not support any structured way of testing.

Recovery Testing: Recovery testing is used to define how well an application can recover from crashes, hardware failure, and other problems. The purpose of recovery testing is to verify the system's ability to recover from testing points of failure.

Static Testing: Static testing is a software testing technique by which we can check the defects in software without actually executing it. Static testing is done to avoid errors in the early stage of the development as it is easier to find failure in the early stages. Static testing used to detect the mistakes that may not found in dynamic testing. **Why static testing?**

Static testing helps to find the error in the early stages. With the help of static testing, this will reduce the development timescales. It reduces the testing cost and time. Static testing also used for development productivity.

Component Testing: Component Testing is also a type of software testing in which testing is performed on each component separately without integrating with other parts. Component testing is also a type of black-box testing. Component testing also referred to as Unit testing, program testing, or module testing.

Grey Box Testing: Grey Box Testing defined as a combination of both white box and black-box testing. Grey Box testing is a testing technique which performed with limited information about the internal functionality of the system.



What are the functional testing tools?

The functional testing can also be executed by various apart from manual testing. These tools simplify the process of testing and help to get accurate and useful results.

It is one of the significant and top-priority based techniques which were decided and specified before the development process.

The tools used for functional testing are:

Tools	Features/Characteristics
Sahi	It is an open-source and automation testing tool, released under Apache License open source license, used for testing of the web application. Sahi is written in Java and JavaScript and considered for most of the testing techniques. It runs as a proxy server; it is browser-independent.
SoapUI	It is an open-source functional testing tool, used for web application testing. It is simple and easy to design. It supports multiple environments, i.e., at any instance, the target environment may be set up.
Watir	Watir, is an abbreviated form of web application testing in ruby, is an open-source tool for automating web browser. It uses a ruby scripting language, which is concise and easy to use. Watir supports multiple browsers on various platform.
Selenium	The open-source tool, used for functional testing on both web application and applications of the desktop. It automates browsers and web application for testing purpose. It gives the flexibility to customize the automated test case Provides the advantage of writing test scripts, as per the requirements, using web driver.
Canoo Web Test	An open-source tool for performing functional testing of the web application. Platform independent Easy and fast Easy to extend to meet growing and incoming requirements.
Cucumber	Cucumber is an open-source testing tool written in Ruby language. This tool works best for test-driven development. It is used to test many other languages like java, c#, and python. Cucumber for testing using some programming.

Advantages of functional testing are:

- It produces a defect-free product.
- It ensures that the customer is satisfied.
- It ensures that all requirements met.
- It ensures the proper working of all the functionality of an application/software/product.
- It ensures that the software/ product work as expected.
- It ensures security and safety.
- It improves the quality of the product.

Example: Here, we are giving an example of banking software. In a bank when money transferred from bank A to bank B. And the bank B does not receive the correct amount, the fee is applied, or the money not converted into the correct currency, or incorrect transfer or bank A does not receive statement advice from bank B that the payment has received. These issues are critical and can be avoided by proper functional testing.

Disadvantages of functional testing are:

Functional testing can miss a critical and logical error in the system.

This testing is not a guarantee of the software to go live.

The possibility of conducting redundant testing is high in functional testing.

We can conclude that to build a strong foundation of a top-class software product, functional testing is essential. It acts as a foundation of the structure, and it is a crucial part of every test routine.

Non-Functional Testing

Non-functional testing is a type of software testing to test non-functional parameters such as reliability, load test, performance and accountability of the software. The primary purpose of non-functional testing is to test the reading speed of the software system as per non-functional parameters. The parameters of non-functional testing are never tested before the functional testing.

Non-functional testing is also very important as functional testing because it plays a crucial role in customer satisfaction.

For example, non-functional testing would be to test how many people can work simultaneously on any software.

Why Non-Functional Testing

Functional and Non-functional testing both are mandatory for newly developed software. Functional testing checks the correctness of internal functions while Non-Functional testing checks the ability to work in an external environment.

It sets the way for software installation, setup, and execution. The measurement and metrics used for internal research and development are collected and produced under non-functional testing.

Non-functional testing gives detailed knowledge of product behaviour and used technologies. It helps in reducing the risk of production and associated costs of the software.

Parameters to be tested under Non-Functional Testing



Performance Testing. Performance Testing eliminates the reason behind the slow and limited performance of the software. Reading speed of the software should be as fast as possible. For Performance Testing, a well-

structured and clear specification about expected speed must be defined. Otherwise, the outcome of the test (Success or Failure) will not be obvious.

Load Testing. Load testing involves testing the system's loading capacity. Loading capacity means more and more people can work on the system simultaneously.

Security Testing. Security testing is used to detect the security flaws of the software application. The testing is done via investigating system architecture and the mindset of an attacker. Test cases are conducted by finding areas of code where an attack is most likely to happen.

Portability Testing. The portability testing of the software is used to verify whether the system can run on different operating systems without occurring any bug. This test also tests the working of software when there is a same operating system but different hardware.

Accountability Testing. Accountability test is done to check whether the system is operating correctly or not. A function should give the same result for which it has been created. If the system gives expected output, it gets passed in the test otherwise failed.

Reliability Testing. Reliability test assumes that whether the software system is running without fail under specified conditions or not. The system must be run for a specific time and number of processes. If the system is failed under these specified conditions, reliability test will be failed.

Efficiency Testing. Efficiency test examines the number of resources needed to develop a software system, and how many of these were used. It also includes the test of these three points.

Customer's requirements must be satisfied by the software system.

A software system should achieve customer specifications.

Enough efforts should be made to develop a software system.

Advantages of Non-functional testing.

It provides a higher level of security. Security is a fundamental feature due to which system is protected from cyber-attacks.

It ensures the loading capability of the system so that any number of users can use it simultaneously.

It improves the performance of the system.

Test cases are never changed so do not need to write them more than once.

Overall time consumption is less as compared to other testing processes.

Disadvantages of Non-Functional Testing

Every time the software is updated, non-functional tests are performed again.

Due to software updates, people have to pay to re-examine the software; thus software becomes very expensive.

Levels of Testing

I. Unit Testing.

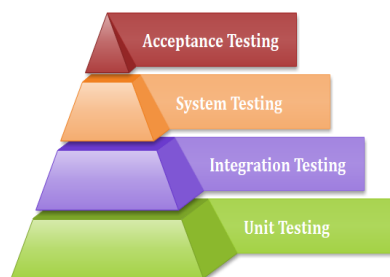
Unit testing involves the testing of each unit or an individual component of the software application. It is the first level of functional testing. The aim behind unit testing is to validate unit components with its performance.

A unit is a single testable part of a software system and tested during the development phase of the application software. The purpose of unit testing is to test the correctness of isolated code. A unit component is an individual function or code of the application. White box testing approach used for unit testing and usually done by the developers.

Whenever the application is ready and given to the Test engineer, he/she will start checking every component of the module or module of the application independently or one by one, and this process is known as Unit testing or components testing.

Why Unit Testing?

In a testing level hierarchy, unit testing is the first level of testing done before integration and other remaining levels of the testing. It uses modules for the testing process which reduces the dependency of waiting for Unit testing frameworks, stubs, drivers and mock objects are used for assistance in unit testing.



Generally, the software goes under four level of testing: Unit Testing, Integration Testing, System Testing, and Acceptance Testing but sometimes due to time consumption software testers does minimal unit testing but skipping of unit testing may lead to higher defects during Integration Testing, System Testing, and Acceptance Testing or even during Beta Testing which takes place after the completion of software application.

Some crucial reasons are listed below:

Unit testing helps tester and developers to understand the base of code that makes them able to change defect causing code quickly.

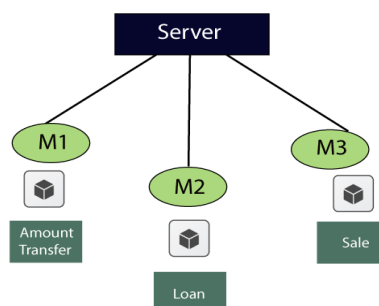
Unit testing helps in the documentation.

Unit testing fixes defects very early in the development phase that's why there is a possibility to occur a smaller number of defects in upcoming testing levels.

It helps with code reusability by migrating code and test cases.

Example of Unit testing

Let us see one sample example for a better understanding of the concept of unit testing:



For the amount transfer, requirements are as follows:

1.	Amount transfer
1.1	From account number (FAN)→ Text Box
1.1.1	FAN→ accept only 4 digit
1.2	To account no (TAN)→ Text Box

1.2.1	TAN→ Accept only 4 digit
1.3	Amount→ Text Box
1.3.1	Amount → Accept maximum 4 digit
1.4	Transfer→ Button
1.4.1	Transfer → Enabled
1.5	Cancel→ Button
1.5.1	Cancel→ Enabled

Below are the application access details, which is given by the customer

URL→ login Page

Username/password/OK → home page

To reach Amount transfer module follow the below

Loans → sales → Amount transfer

While performing unit testing, we should follow some rules, which are as follows:

To start unit testing, at least we should have one module.

Test for positive values

Test for negative values

No over testing

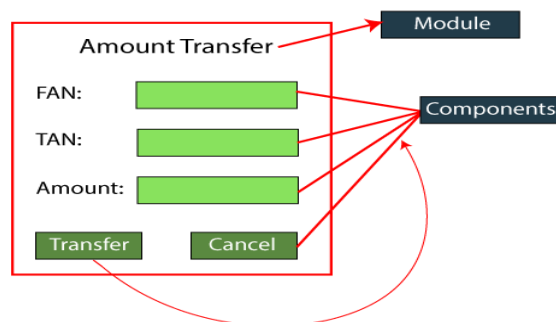
No assumption required

When we feel that the **maximum test coverage** is achieved, we will stop the testing. Now, we will start performing the unit testing on the different components such as

From account number(FAN)

To account number(TAN)

Amount



For the FAN components

Values	Description
1234	accept
4311	Error message→ account valid or not
blank	Error message→ enter some values
5 digit/ 3 digit	Error message→ accept only 4 digit
Alphanumeric	Error message → accept only digit
Blocked account no	Error message
Copy and paste the value	Error message→ type the value
Same as FAN and TAN	Error message

For the TAN component

Provide the values just like we did in From account number (FAN) components

For Amount component

Provide the values just like we did in FAN and TAN components.

For Transfer component

Enter valid FAN value

Enter valid TAN value

Enter the correct value of Amount

Click on the Transfer button → amount transfer successfully (confirmation message)

For Cancel Component

Enter the values of FAN, TAN, and amount.

Click on the Cancel button → all data should be cleared.

Unit Testing Tools

We have various types of unit testing tools available in the market, which are as follows:

NUnit

JUnit

PHPUnit

Parasoft Jtest

EMMA

Unit Testing Techniques:

Unit testing uses all white box testing techniques as it uses the code of software application:

Data flow Testing

Control Flow Testing

Branch Coverage Testing

Statement Coverage Testing

Decision Coverage Testing

How to achieve the best result via Unit testing?

Unit testing can give best results without getting confused and increase complexity by following the steps listed below:

Test cases must be independent because if there is any change or enhancement in requirement, the test cases will not be affected.

Naming conventions for unit test cases must be clear and consistent.

During unit testing, the identified bugs must be fixed before jump on next phase of the SDLC.

Only one code should be tested at one time.

Adopt test cases with the writing of the code, if not doing so, the number of execution paths will be increased.

If there are changes in the code of any module, ensure the corresponding unit test is available or not for that module.

Advantages and disadvantages of unit testing

The pros and cons of unit testing are as follows:

Advantages

Unit testing uses module approach due to that any part can be tested without waiting for completion of another parts testing.

The developing team focuses on the provided functionality of the unit and how functionality should look in unit test suits to understand the unit API.

Unit testing allows the developer to refactor code after a number of days and ensure the module still working without any defect.

Disadvantages

It cannot identify integration or broad level error as it works on units of the code.

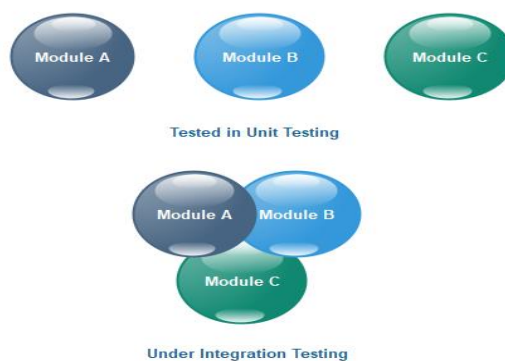
In the unit testing, evaluation of all execution paths is not possible, so unit testing is not able to catch each and every error in a program.

It is best suitable for conjunction with other testing activities.

II. Integration Testing.

Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.



Why Integration Testing

Although all modules of software application already tested in unit testing, errors still exist due to the following reasons:

Each module is designed by individual software developer whose programming logic may differ from developers of other modules so; integration testing becomes essential to determine the working of software modules.

To check the interaction of software modules with the database whether it is an erroneous or not.

Requirements can be changed or enhanced at the time of module development. These new requirements may not be tested at the level of unit testing hence integration testing becomes mandatory.

Incompatibility between modules of software could create errors.

To test hardware's compatibility with software.

If exception handling is inadequate between modules, it can create bugs.

Integration Testing Techniques

Any testing technique (Blackbox, Whitebox, and Greybox) can be used for Integration Testing; some are listed below:

Black Box Testing

State Transition technique

Decision Table Technique

Boundary Value Analysis

All-pairs Testing

Cause and Effect Graph

Equivalence Partitioning

Error Guessing

White Box Testing

Data flow testing

Control Flow Testing

Branch Coverage Testing

Decision Coverage Testing

Grey Box Testing

Methodologies of Integration Testing

There are two basic methods for Integration Testing:

Big Bang

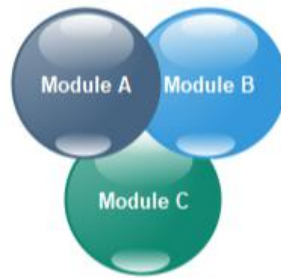
Incremental

Now, let's understand these approach.

Big Bang Method

In this approach, testing is done via integration of all modules at once. It is convenient for small software systems, if used for large software systems identification of defects is difficult.

Since this testing can be done after completion of all modules due to that testing team has less time for execution of this process so that internally linked interfaces and high-risk critical modules can be missed easily.



Advantages: It is convenient for small size software systems.

Disadvantages: Identification of defect is difficult. Small modules missed easily. Time provided for testing is very less.

Incremental Approach

In the Incremental Approach, modules are added in ascending order one by one or according to need. The selected modules must be logically related. Generally, two or more than two modules are added and tested to determine the correctness of functions. The process continues until the successful testing of all the modules.

Incremental Approach is carried out by further methods:

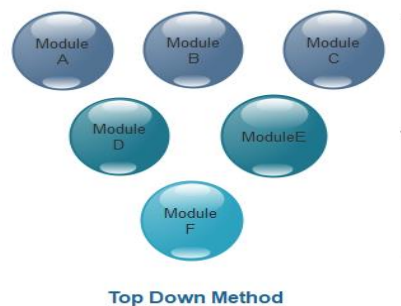
Top Down

Bottom Up

Hybrid Testing

Top-Down Method

The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules. Major design flaws can be detected and fixed early because critical modules tested first.



Advantages:

An early prototype is possible.

Critical Modules are tested first so that fewer chances of defects.

Disadvantages:

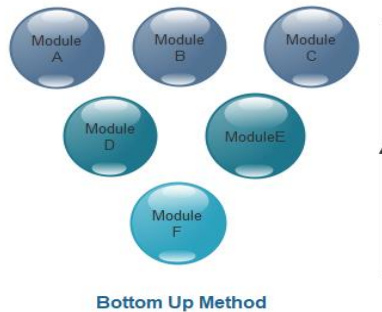
Due to the high number of stubs, it gets quite complicated.

Identification of defect is difficult.

Lower level modules are tested inadequately.

Bottom Up Method

The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules. Top level critical modules are tested at last, so it may cause a defect.



Advantages:

Identification of defect is easy.

Do not need to wait for the development of all the modules as It saves time.

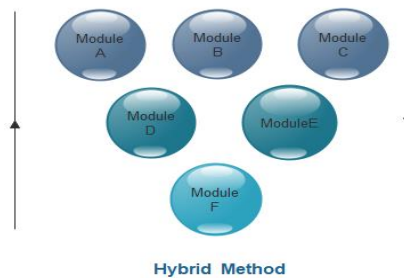
Disadvantages:

Critical modules are tested last due to which the defects can occur.

There is no possibility of an early prototype.

Hybrid Testing Method

In this approach, both Top-Down and Bottom-Up approaches are combined for testing. In this process, top-level modules are tested with lower level modules and lower level modules tested with high-level modules simultaneously. There is less possibility of occurrence of defect because each module interface is tested.



Advantages:

The hybrid method provides features of both Bottom Up and Top Down methods.

It is most time reducing method.

It provides complete testing of all modules.

Disadvantages:

This method needs a higher level of concentration as the process carried out in both directions simultaneously.

Complicated method.

Guidelines for Integration Testing

First, determine the test case strategy through which executable test cases can be prepared according to test data.

Examine the structure and architecture of the application and identify the crucial modules to test them first.

Design test cases to verify each interface in detail.

Choose input data for test case execution. Input data plays a significant role in testing.

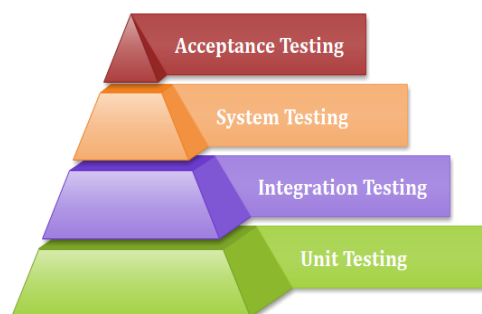
Fix defects and retest.

III. 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.

There is four levels of software testing: *unit testing*, *integration testing*, *system testing* and *acceptance testing*, all are used for the testing purpose. Unit Testing used to test a single software; Integration Testing used to test a group of units of software, System Testing used to test a whole system and Acceptance Testing used to test the acceptability of business requirements. Here we are discussing system testing which is the third level of testing levels.

Hierarchy of Testing Levels



There are mainly two widely used methods for software testing, one is White box testing which uses internal coding to design test cases and another is black box testing which uses GUI or user perspective to develop test cases.

White box testing

Black box testing

[A GUI (graphical user interface) is a system of interactive visual components for computer software. A GUI displays objects that convey information, and represent actions that can be taken by the user. The objects change color, size, or visibility when the user interacts with them. GUI objects include icons, cursors, and buttons.]

System testing falls under Black box testing as it includes testing of the external working of the software. Testing follows user's perspective to identify minor defects.

System Testing includes the following steps.

Verification of input functions of the application to test whether it is producing the expected output or not.

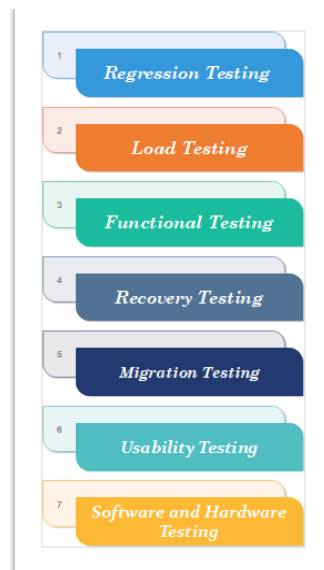
Testing of integrated software by including external peripherals to check the interaction of various components with each other.

Testing of the whole system for End to End testing.

Behavior testing of the application via a user's experience

Types of System Testing

System testing is divided into more than 50 types, but software testing companies typically use some of them. These are listed below:



Regression Testing

Regression testing is performed under system testing to confirm and identify that if there's any defect in the system due to modification in any other part of the system. It makes sure, any changes done during the development process have not introduced a new defect and also gives assurance; old defects will not exist on the addition of new software over the time.

Load Testing

Load testing is performed under system testing to clarify whether the system can work under real-time loads or not.

Functional Testing

Functional testing of a system is performed to find if there's any missing function in the system. Tester makes a list of vital functions that should be in the system and can be added during functional testing and should improve quality of the system.

Recovery Testing

Recovery testing of a system is performed under system testing to confirm reliability, trustworthiness, accountability of the system and all are lying on recouping skills of the system. It should be able to recover from all the possible system crashes successfully.

Migration Testing

Migration testing is performed to ensure that if the system needs to be modified in new infrastructure so it should be modified without any issue.

Usability Testing

The purpose of this testing to make sure that the system is well familiar with the user and it meets its objective for what it supposed to do.

Software and Hardware Testing

This testing of the system intends to check hardware and software compatibility. The hardware configuration must be compatible with the software to run it without any issue. Compatibility provides flexibility by providing interactions between hardware and software.

Importance of System Testing.

System Testing gives hundred percent assurance of system performance as it covers end to end function of the system.

It includes testing of System software architecture and business requirements.

It helps in mitigating live issues and bugs even after production.

System testing uses both existing system and a new system to feed same data in both and then compare the differences in functionalities of added and existing functions so, the user can understand benefits of new added functions of the system.

Alpha and Beta Testing

Alpha Testing

Alpha testing is conducted in the organization and tested by a representative group of end-users at the developer's side and sometimes by an independent team of testers.

Alpha testing is simulated or real operational testing at an in-house site. It comes after the unit testing, integration testing, etc. Alpha testing used after all the testing are executed.

It can be a white box, or Black-box testing depends on the requirements - particular lab environment and simulation of the actual environment required for this testing.

Alpha testing follows the following process:

Requirement Review: Review the design of the specification and functional requirement

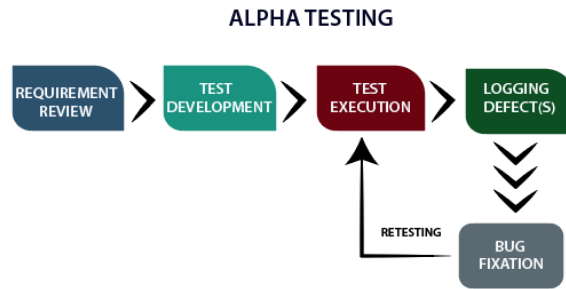
Test Development: Test development is base on the outcome of the requirement review. Develop the test cases and test plan.

Test case design: Execute the test plan and test cases.

Logging Defects: Logging the identified and detected bug found in the application.

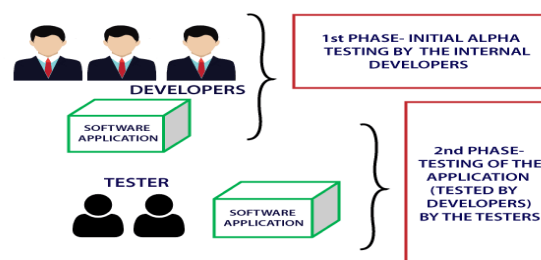
Bug Fixation: When all the bugs are identified and logged, then there is a need to fix the bug.

Retesting: When all the issues are solved, and fixed retesting is done.



Phases of alpha testing

Alpha testing ensures that the software performs flawlessly and does not impact the reputation of the organization; the company implements final testing in the form of alpha testing. This testing executed into two phases.

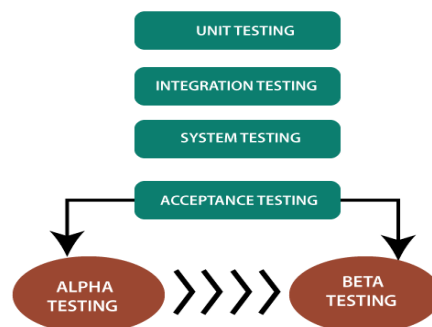


There are two phases of alpha testing.

First Phase: In-house developers of software engineers do the first phase of testing. In this phase, the tester used hardware debugger or hardware aided debugger to catches the bugs quickly. During the alpha testing, a tester finds a lot of bugs, crashes, missing features, and docs.

Second Phase: The second phase involves the quality assurance staff performs the alpha testing by involving black box and white box techniques.

When to perform Alpha Testing?



Alpha testing is user acceptance testing. Alpha testing performed once the product has gone through stages of testing and prepared for release. It is executing before beta testing, which is also a part of acceptance testing and can define as field testing. During this testing, we can make changes in the software to improve its quality and functionality. Alpha testing done from the developer's site where independent developers can monitor and record user experience and make necessary changes to enhance the performance.

Reasons for Alpha Testing

Alpha testing is the final stage of the testing. Alpha testing is an essential and popular testing technique that helps the team to deliver quality and useful software. This testing performed before the release of the product. Alpha testing can define as the first round of independent testing that ensures that the software run as per the requirement plan.

Reasons for alpha testing are:

Refines the software product by finding and rectifying bugs that weren't discovered through previous tests.

Alpha testing allows the team to test the software in a real-world environment.

One of the reasons to do alpha testing is to ensure the success of the software product.

Alpha testing validates the quality, functionality of the software, and effectiveness of the software before it released in the real world.

Features of Alpha Testing

Alpha testing is a type of acceptance testing.

Alpha testing is happening at the stage of the completion of the software product.

Alpha testing is in the labs where we provide a specific and controlled environment.

Alpha testing is in-house testing, which is performed by the internal developers and testers within the organization.

There is not any involvement of the public.

Alpha testing helps to gain confidence in the user acceptance of the software product.

With the help of black box and white box technique, we can achieve the alpha testing.

Alpha testing ensures the maximum possible quality of the software before releasing it to market or client for beta testing.

Developers perform alpha testing at developer's site; it enables the developer to record the error with the ease to resolve found bugs quickly.

Alpha testing is doing after the unit testing, integration testing, system testing but before the beta testing.

Alpha testing is for testing the software application, products, and projects.

Advantages of alpha testing are:

One of the benefits of alpha testing is it reduces the delivery time of the project.

It provides a complete test plan and test cases.

Free the team member for another project.

Every feedback helps to improve software quality.

It provides a better observation of the software's reliability and accountability.

Disadvantages of alpha testing are:

Alpha testing does not involve in-depth testing of the software.

The difference between the tester's tests the data for testing the software and the customer's data from their perspective may result in the discrepancy in the software functioning.

The lab environment is used to simulate the real environment. But still, the lab cannot furnish all the requirement of the real environment such as multiple conditions, factors, and circumstances.

Wrap Up:

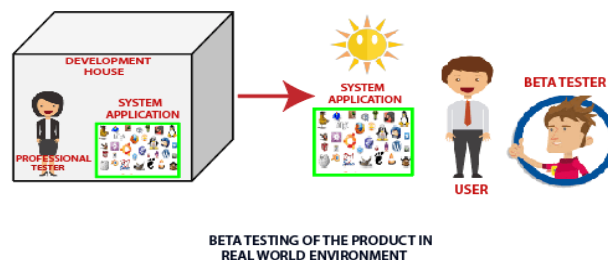
Every software product needs to undergo a vital methodology before going into a highly competitive market. Alpha testing is one of the vital testing. It needs to be considered by going through the functionality of the software and achieve the confidence in its user acceptance for the real environment, before releasing it into the market.

Recall: If we compare the various activity performed to develop ideal software, we will find the importance of software testing similar to that of the software development process. Testing is one of those activities which ensure the accuracy of the development process while validating its functionality and performance.

Beta Testing

Beta testing is a type of User Acceptance Testing among the most crucial testing, which performed before the release of the software. Beta Testing is a type of Field Test. This testing performs at the end of the software testing life cycle. This type of testing can be considered as external user acceptance testing. It is a type of salient testing. Real users perform this testing. This testing executed after the alpha testing. In this the new version, beta testing is released to a limited audience to check the accessibility, usability, and functionality, and more.

Beta testing is the last phase of the testing, which is carried out at the client's or customer's site.



Features of beta testing

Testing of the product performs by the real users of the software application in the real environment. Beta version of the software is released to a restricted number of end-users to obtain the feedback of the product quality. Beta testing reduces the risk of failure and provides the quality of the product through customer validation. It is the final testing before shipping the product to the customers. Beta testing obtains direct feedback from the customers. It helps in testing to test the product in the customer's environment.

Features of beta testing are:

Beta testing used in a real environment at the user's site. Beta testing helps in providing the actual position of the quality.

Testing performed by the client, stakeholder, and end-user.

Beta testing always is done after the alpha testing, and before releasing it into the market.

Beta testing is black-box testing.

Beta testing performs in the absence of tester and the presence of real users

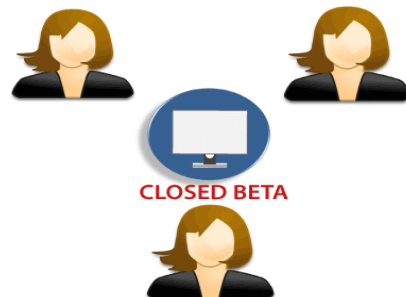
Beta testing is performed after alpha testing and before the release of the final product.

Beta testing generally is done for testing software products like utilities, operating systems, and applications, etc.

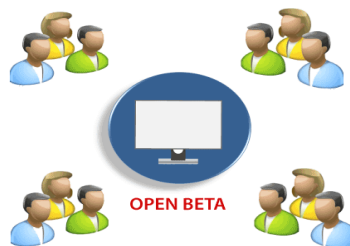
Beta version of the software

The beta version of the software is delivered to a restricted number of users to accept their feedback and suggestions on quality improvement. Hence, there are two types of beta version:

Closed beta version: Closed beta version, also known as a private beta, it is released to a group of selected and invited people. Those people will test the software and evaluate their features and specifications. This beta version represents the software which is capable of delivering value, but it is not ready to be used by everyone. Because it shows the issues like lack of documentation or missing vital features.



Open beta version: Open beta is also known as a public beta. The open beta is opened to the public. Any user as a tester can assess the beta version to provide the relevant feedback and reviews. Open beta version improves the quality of the final release. This version helps to find the various undetected errors and issues.



Lifecycle of Beta Testing

A group of end-users performs beta testing. This process can't execute without any strategy or test plan. Before the testers, the end-user executes this type of testing.

The process of beta testing follows the following steps:

Planning: Like another testing process, beta testing also supports proper planning. In this stage, the team prepares a testing strategy and defines the goal of testing. In this case, the team establishes the need of users for testing, duration, and necessary details related to the process.

Participant Recruitment: This is the second stage of the beta process in which the team recruits a group of selected end-users for testing. This group can change as per the requirement of the organization and the product.

Product Launch: When a team of users (testers) recruited. The beta version of the product is launched or installed at the client or user side, and users will test the product for quality assurance.

Collect and Evaluate Feedback: When the testing finished, developers will collect the feedback provided by the testers and evaluate it. In the end, based on the feedback, issues, and bugs are fixed and resolved by the responsible individual team.

Closure: When all the problems fixed and the organization meets the exit criteria, beta testing achieved, and the rewards offered to the testing team.

Types of Beta Testing

Beta testing has six types. Each type has different aspects of the software. All these help developers to improve the quality of the software and allow them to deliver a product that offers excellent user experience. Here are the different types of beta testing:

Open Beta Testing: Open beta testing involves testing the software product by a large number of people before the final release. The organization decides to make a software product open to the public before releasing the product. Open Beta includes the extensive participation of the public to use and evaluate software product accordingly.

Users report the bug to the organization, along with a suggestion to improve the quality of the software.

Closed Beta Testing: Opposite to the open beta testing. Closed beta testing performed by the selective and limited number of persons. The organization recruits these. In this testing software product is not open to the public.

Traditional Beta Testing: In this testing, a software product delivered to the target market, and the feedback from the users collected. This type of testing assistance the beta testing, the quality of the software is improved, and developers can make the changes.

Public Beta Testing: This type of testing is similar to open testing. Public beta testing also allows the product is delivering to the end-users worldwide, with the aid of various online channels available in the world. From this, the feedback and evaluated data also collected and based on the requirement changes, and the development team implements modifications.

Technical Beta Testing: Technical beta testing is also an essential type of beta testing. This testing involves delivering the software product to the internal groups of the organization. However, the data and feedback provided by the employees of the organization.

Focused Beta Testing: This type of testing focused on monitoring and evaluating a specific feature or component of the software. In focused beta testing, the software released to the market and user's experience assessed and collected to make the required changes.

Post-Release Beta Testing: In this testing, the product delivered to the market for the use of the end-users. Their feedback, reactions, and experience are collect for the future release of the software.

When to perform Beta Testing?

Acceptance testing is the final phase of the testing, which combines both alpha and beta testing to ensure that the product released flawlessly. Beta testing performed at the user's end. This testing always performed after the alpha testing, but before the product released to the market. In this stage, the product is expected to be 90% to 95% completed. Any product undergoing to beta test should be reviewed for the entire checklist before launching it.

Some of them are:

- All the component of the product is ready to start this testing.

- Documentation which is going to end-user should be kept ready - Setup, installation, usage, Uninstallation should be in detail.

- The product management team should review that all the functionality is in good condition.

- Procedure to collect bugs, feedback, etc. should be identified before publishing it.

Stakeholders and participants in the Beta Testing

The product management, quality management, and user experience teams are the stakeholder in beta testing, and they closely monitor every move of the phase.

The real users who use the product are the participants.

Beta test strategy

Business objective for the product.

Beta test plan

The testing approach followed by participants.

Tools used to detect bugs, measure productivity, collect feedback.

When and how to end this testing phase?

Beta Test plan

A beta test plan can be written in many ways,

Objective: We should have to mention the aim of the project why there is a need for beta testing even after performing the internal testing.

Scope: In this plan, we should mention the areas to be tested or not.

Test Approach: We should have to mention clearly that the testing is in the deep, what to focus on - functionality, UI, response, etc.

Schedule: We have to specify, clearly the start and ending date with time, number of cycles, and duration per cycle.

Tools: Bug logging tools and the usage of the machines should identify.

Budget: Incentive of the bugs based on the severity.

Feedback: Collecting feedback and evaluating methods.

Identify and review the entry and exit criteria.

Entry criteria for Beta Testing

Sign off the document from alpha testing.

Beta version of the software should ready.

The environment should be ready to release the software application to the public.

To capture the real-time faults environment should be ready.

Exit criteria for Beta Testing

All the major and minor issues resolved.

The feedback report should prepare.

The delivery of beta test summary report.

Advantages of Beta Testing

Beta testing performed at the end of the software testing lifecycle. Beta testing offers numerous benefits to testers, software developer, as well as the users. In the assistance of this type of testing, it enables developers, testers to test the product before its release in the market.

The Beta testing focuses on the customer's satisfaction.

It helps to reduce the risk of product failure via user validations.

Beta testing helps to get direct feedback from users.

It helps to detect the defect and issues in the system, which is overlooked and undetected by the team of software testers.

Beta testing helps the user to install, test, and send feedback regarding the developed software.

Disadvantages of Beta Testing

In this type of testing, a software engineer has no control over the process of the testing, as the users in the real-world environment perform it.

This testing can be a time-consuming process and can delay the final release of the product.

Beta testing does not test the functionality of the software in depth as software still in development.

It is a waste of time and money to work on the feedback of the users who do not use the software themselves properly.

Remarks

Keeping in mind the characteristics of beta testing it can be concluded that beta testing may be considered desirable for the organization. Beta testing provides the feedback of the real-users, which helps improve the software quality before the product released in the market.

Debugging

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. As a software engineer, you are often confronted with a “symptomatic” indication of a software problem as you evaluate the results of a test. That is, the external manifestation of the error and its internal cause may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

Debugging process

Debugging is not testing but often occurs as a consequence of testing. The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non-corresponding data are a symptom of an underlying cause as yet hidden. *The debugging process attempts to match symptom with cause, thereby leading to error correction.*

The debugging process will usually have one of two outcomes:

- (1) the cause will be found and corrected or
- (2) the cause will not be found.

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Why is debugging so difficult?

Few characteristics of bugs that provide some clues:

The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components exacerbate this situation.

The symptom may disappear (temporarily) when another error is corrected.

The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).

The symptom may be caused by human error that is not easily traced.

The symptom may be a result of timing problems, rather than processing problems.

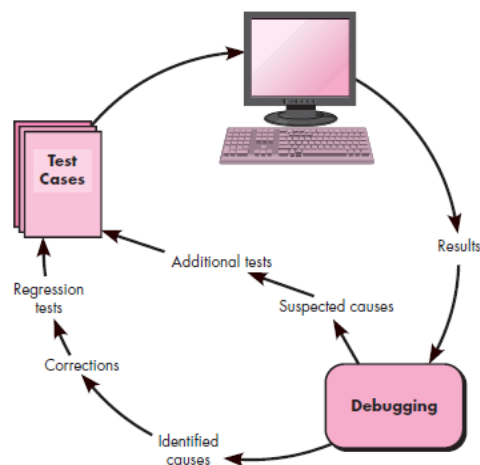
It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, you'll encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces some software developers to fix one error and at the same time introduce two more.

The following figure illustrates the debugging process:



Debugging strategies

Regardless of the approach that is taken, debugging has one overriding objective—to find and correct the cause of a software error or defect. The objective is realized by a combination of systematic evaluation, intuition, and luck.

In general, three debugging strategies have been proposed: **brute force**, **backtracking**, and **cause elimination**. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective. [Automated debugging. Each of these debugging approaches can be supplemented with debugging tools that can provide you with semi-automated support as debugging strategies are attempted.]

Brute force. The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error. Popular because it requires little thought and is the least mentally taxing of the methods, but it is inefficient and generally unsuccessful. You apply brute force debugging methods when all else fails. Using a “let the computer find the error” philosophy, memory/storage dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you’ll find a clue that can lead to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

Brute force methods can be partitioned into at least three categories:

Debugging with a storage dump.

Debugging according to the common suggestion to "scatter print statements throughout your program."

Debugging with automated debugging tools.

The first, debugging with a storage dump is the most inefficient of the brute force methods. Here's why:

It is difficult to establish a correspondence between memory locations and the variables in a source program.

With any program of reasonable complexity, such a memory dump will produce a massive amount of data, most of which is irrelevant.

A memory dump is a static picture of the program, showing the state of the program at only one instant in time; to find errors, you have to study the dynamics of a program (state changes over time).

A memory dump is rarely produced at the exact point of the error, so it doesn't show the program's state at the point of the error. Program actions between the time of the dump and the time of the error can hide/mask the clues you need to find the error.

There aren't adequate methodologies for finding errors by analyzing a memory dump (so many programmers stare, with glazed eyes, wistfully ...

Backtracking. Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

Cause Elimination. The third approach to debugging—cause elimination—is manifested by induction or deduction and introduces the concept of binary partitioning. **Data related to the error occurrence are organized to isolate potential causes.** A “cause hypothesis” is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Correcting the Error

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [1989] suggests three simple questions that you should ask before making the “correction” that removes the cause of a bug:

1. Is the cause of the bug reproduced in another part of the program? In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.

2. What “next bug” might be introduced by the fix I’m about to make? Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.
3. What could we have done to prevent this bug in the first place? This question is the first step toward establishing a statistical software quality assurance approach (Chapter 16). If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

[T. Van Vleck, “Three Questions About Each Bug You Find,” *ACM Software Engineering Notes*, vol. 14, no. 5, July 1989, pp. 62–63.]

Software Testing Strategies

A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses.

In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our only defence against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques [and technical reviews] are helping us to reduce the number of initial errors that are inherent in the code. Similarly, different test methods are beginning to cluster themselves into several distinct approaches and philosophies.

These “approaches and philosophies” are what can be called strategy. The testing methods and techniques that implement the strategy have been discussed. Here we shall present them in summary again.

A Quick Recap

What is it? Software is tested to uncover errors that were made inadvertently as it was designed and constructed. But how do you conduct the tests? Should you develop a formal plan for your tests? Should you test the entire program as a whole or run tests only on a small part of it? Should you rerun tests you’ve already conducted as you add new components to a large system? When should you involve the customer? **These and many other questions are answered when you develop a software testing strategy.**

Who does it? A strategy for software testing is developed by the project manager, software engineers, and testing specialists.

Why is it important? Testing often accounts for more project effort than any other software engineering action. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected. It would therefore seem reasonable to establish a systematic strategy for testing software.

What are the steps? Testing begins “in the small” and progresses “to the large.” By this I mean that early testing focuses on a single component or a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

What is the work product? A Test Specification documents the software team's approach to testing by defining a plan that describes an overall strategy and a procedure that defines specific testing steps and the types of tests that will be conducted.

How do I ensure that I've done it right? By reviewing the Test Specification prior to testing, you can assess the completeness of test cases and testing tasks. An effective test plan and procedure will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

A 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 you 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 you with a template for testing and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.

- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.

- Different testing techniques are appropriate for different software engineering approaches and 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.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy should provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

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 tasks that ensure that software correctly implements a specific function. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [1981] states this another way:

Verification: “*Are we building the product right?*”

Validation: “*Are we building the right product?*”

The definition of V&V encompasses many software quality assurance activities.

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing.

Although testing plays an extremely important role in V&V, many other activities are also necessary. Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, “You can’t test in quality. If it’s not there before you begin testing, it won’t be there when you’re finished testing.” Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

Miller [1977] relates software testing to quality assurance by stating that “the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems.”

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. From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer analyzes, models, and then creates a computer program and its documentation.

Like any builder, the software engineer is proud of the edifice that has been built and looks suspicious at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to “break” the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) destructive. So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than uncovering errors. Unfortunately, errors will be present. And, if the software engineer doesn’t find them, the customer will!

There are often a number of misconceptions: (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.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behaviour for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.

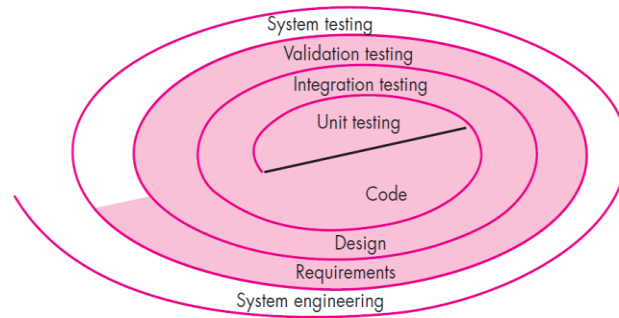
The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

However, you don’t turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

The ITG is part of the software development project team in the sense that it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering organisation.

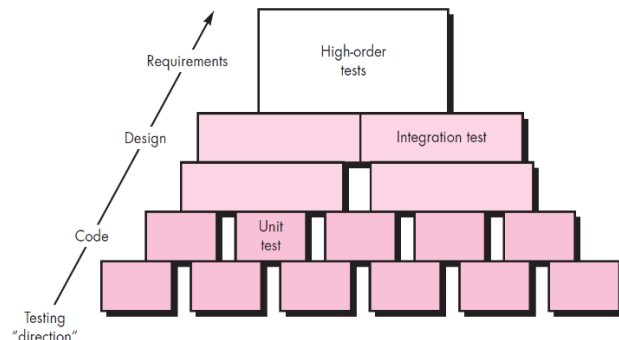
Software Testing Strategy: The Big Picture

The software process may be viewed as the spiral illustrated in following Figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behaviour, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counter-clockwise) along streamlines that decrease the level of abstraction on each turn. **Figure 6.1. Testing strategy.**



A strategy for software testing may also be viewed in the context of the spiral. Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter validation testing, where requirements established as part of requirements modelling are validated against the software that has been constructed. Finally, you arrive at system testing, where the software and other system elements are tested as a whole. To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in **Figure 6.2. Software testing steps**. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.



Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high-order tests is conducted. Validation criteria (established during requirements analysis) must be evaluated. Validation testing provides final assurance that software meets all informational, functional, behavioural, and performance requirements.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

Criteria for Completion of Testing

A classic question arises every time software testing is discussed: "When are we done testing-how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: “You’re never done testing; the burden simply shifts from you (the software engineer) to the end user.” Every time the user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities. Another response (somewhat cynical but nonetheless accurate) is: “You’re done testing when you run out of time or you run out of money.”

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The cleanroom software engineering approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population. Others advocate using statistical modelling and software reliability theory to predict the completeness of testing.

By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: “When are we done testing?” There is little debate that further work remains to be done before quantitative rules for testing can be established, but the empirical approaches that currently exist are considerably better than raw intuition.

Strategic Issues

There is a systematic strategy for software testing. But even the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb [1995] argues that a software testing strategy will succeed when software testers:

Specify product requirements in a quantifiable manner long before testing commences. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability. These should be specified in a way that is measurable so that testing results are unambiguous.

State testing objectives explicitly. The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, meantime-to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours should be stated within the test plan.

Understand the users of the software and develop a profile for each user category. Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

Develop a testing plan that emphasizes “rapid cycle testing.” Gilb [1995] recommends that a software team “learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field ‘trialable,’ increments of functionality and/or quality improvement.” The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Build “robust” software that is designed to test itself. Software should be designed in a manner that uses anti-bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.

Use effective technical reviews as a filter prior to testing. Technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high quality software.

Conduct technical reviews to assess the test strategy and test cases themselves. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

Test Strategies for Conventional Software

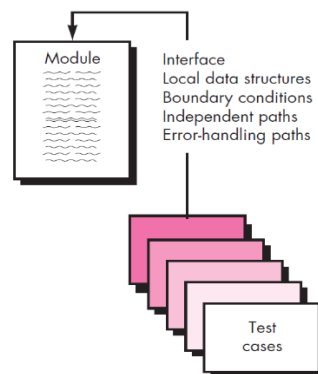
There are many strategies that can be used to test software. At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors. This approach,

although appealing, simply does not work. It will result in buggy software that disappoints all stakeholders. 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.

Unfortunately, some software developers hesitate to use it. What to do? 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, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design-the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. **Figure 6.3. Unit Test.**



The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. 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-test considerations. Unit tests are illustrated schematically in Figure 6.3. 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. And finally, all error-handling paths are tested.

Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the n th element of an n -dimensional array is processed, when the i th repetition of a loop with i passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon [1975] calls this approach anti-bugging. Unfortunately, there is a tendency to incorporate error handling into software and then never test it. A true story may serve to illustrate:

A computer-aided design system was developed under contract. In one transaction processing module, a practical joker placed the following error handling message after a series of conditional tests that

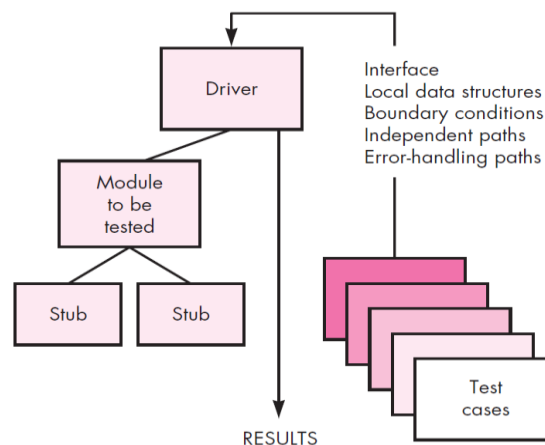
invoked various control flow branches: ERROR! THERE IS NO WAY YOU CAN GET HERE. This “error message” was uncovered by a customer during user training!

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-test procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. The unit test environment is illustrated in Figure 6.4. In most applications a driver is nothing more than a “main program” that accepts test case data, passes such data to the component (to be tested), and prints relevant results.

Figure 6.4. Unit-test environment.



Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or “dummy subprogram” uses the subordinate module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing. Drivers and stubs represent testing “overhead.” That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with “simple” overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Integration Testing

A neophyte (a beginner or a novice) in the software world might ask a seemingly legitimate question once all modules have been unit tested: “If they all work individually, why do you doubt that they’ll work when we put them together?” The problem, of course, is “putting them together”—interfacing.

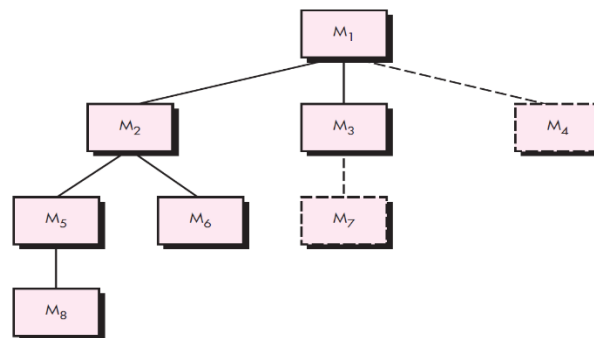
Data can be lost across an interface; one component can have an inadvertent, adverse effect on another; sub-functions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non-incremental integration; that is, to construct the program using a “big bang” approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the paragraphs that follow, a number of different incremental integration strategies are discussed.

Top-down integration. 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. **Figure 6.5.** *Top-down integration.*



Referring to Figure 6.5, depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.

From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows. The integration process is performed in a series of five steps:

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

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

Tests are conducted as each component is integrated.

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

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.

The top-down integration strategy verifies major control or decision points early in the test process. In a “well-factored” program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

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.

The first approach (delay tests until stubs are replaced by actual modules) can cause you to lose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex. The third approach, called bottom-up integration is discussed in the paragraphs that follow.

Bottom-up integration. Bottom-up integration testing, as its name implies, 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:

Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

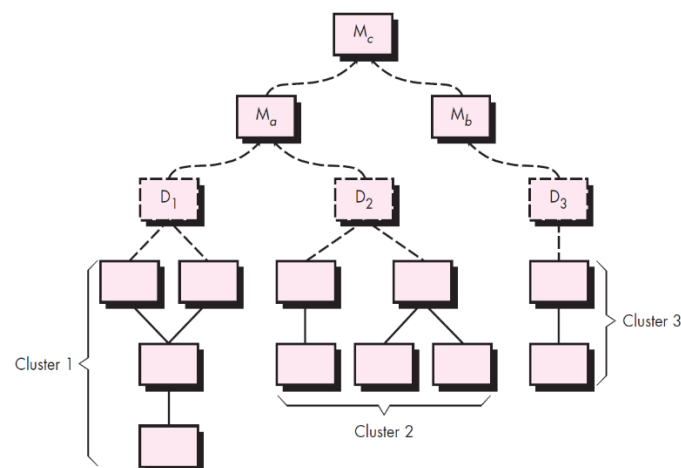
A driver (a control program for testing) is written to coordinate test case input and output.

The cluster is tested.

Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in Figure 6.6. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a . Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a . Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b . Both M_a and M_b will ultimately be integrated with component M_c , and so forth.

Figure 6.6. Bottom-up integration



As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression testing. Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, 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.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. 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. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison. The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.

- Additional tests that focus on software functions that are likely to be affected by the change.

- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Smoke testing. Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis. In essence, the smoke-testing approach encompasses the following activities:

- 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 that are required to implement one or more product functions.

- A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover “showstopper” errors that have the highest likelihood of throwing the software project behind schedule.

- The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [1996] describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

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

- Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.

- 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.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with “new software increments”—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.

- Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

Strategic options. There has been much discussion about the relative advantages and disadvantages of top-down versus bottom-up integration testing. In general, the advantages of one strategy tend to result in disadvantages for the other strategy.

The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them. Problems associated with stubs may be offset by the advantage of testing major control functions early. The major disadvantage of bottom-up integration is that “the program as an entity does not exist until the last module is added” [Mye 1979]. This drawback is tempered by easier test case design and a lack of stubs.

Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called sandwich testing) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

As integration testing is conducted, the tester should identify critical modules. A critical module has one or more of the following characteristics: (1) addresses several software requirements, (2) has a high level of control (resides relatively high in the program structure), (3) is complex or error prone, or (4) has definite performance requirements. Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

Integration test work products. An overall plan for integration of the software and a description of specific tests is documented in a Test Specification. This work product incorporates a test plan and a test procedure and becomes part of the software configuration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for the *SafeHome* security system might be divided into the following test phases:

User interaction (command input and output, display representation, error processing and representation)

Sensor processing (acquisition of sensor output, determination of sensor conditions, actions required as a consequence of conditions)

Communications functions (ability to communicate with central monitoring station)

Alarm processing (tests of software actions that occur when an alarm is encountered)

Each of these integration test phases delineates a broad functional category within the software and generally can be related to a specific domain within the software architecture. Therefore, program builds (groups of modules) are created to correspond to each phase. The following criteria and corresponding tests are applied for all test phases:

Interface integrity. Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

Functional validity. Tests designed to uncover functional errors are conducted. Information content. Tests designed to uncover errors associated with local or global data structures are conducted.

Performance. Tests designed to verify performance bounds established during software design are conducted.

A schedule for integration, the development of overhead software, and related topics are also discussed as part of the test plan. Start and end dates for each phase are established and “availability windows” for unit-tested modules are defined. A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic simulators, and special test tools or techniques are a few of many topics that may also be discussed.

The detailed testing procedure that is required to accomplish the test plan is described next. The order of integration and corresponding tests at each integration step are described. A listing of all test cases (annotated for subsequent reference) and expected results are also included.

A history of actual test results, problems, or peculiarities is recorded in a Test Report that can be appended to the Test Specification, if desired. Information contained in this section can be vital during software maintenance. Appropriate references and appendixes are also presented.

Like all other elements of a software configuration, the test specification format may be tailored to the local needs of a software engineering organization. It is important to note, however, that an integration strategy (contained in a test plan) and testing details (described in a test procedure) are essential ingredients and must appear.

Test Strategies for Object-Oriented Software

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of object-oriented software changes both testing strategy and testing tactics.

Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

You can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class. To illustrate, consider a class hierarchy in which an operation X is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation X, but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation X is used varies in subtle ways, it is necessary to test operation X in the context of each of the subclasses. This means that testing operation

X in a stand-alone fashion (the conventional unit-testing approach) is usually ineffective in the object-oriented context.

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

Integration Testing in the OO Context

Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the “direct and indirect interactions of the components that make up the class”.

There are two different strategies for integration testing of OO systems. The first, thread-based testing, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) server classes. After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

The use of drivers and stubs also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

Test Strategies for WebApps

The strategy for webapp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The Following steps summarize the approach:

The content model for the webapp is reviewed to uncover errors.

The interface model is reviewed to ensure that all use cases can be accommodated.

The design model for the webapp is reviewed to uncover navigation errors.

The user interface is tested to uncover errors in presentation and/or navigation mechanics.

Each functional component is unit tested.

Navigation throughout the architecture is tested.

The webapp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.

Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.

Performance tests are conducted.

The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

Because many WebApps evolve continuously, the testing process is an ongoing activity, conducted by support staff who use regression tests derived from the tests developed when the WebApp was first engineered.

Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between conventional software, object-oriented software, and WebApps disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a *Software Requirements Specification* has been developed, it describes all user-visible attributes of the software and contains a *Validation Criteria* section that forms the basis for a validation-testing approach.

Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct,

and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created. Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

Configuration Review

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review is sometimes called an audit.

Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

The *alpha test* is conducted at the developer’s site by a representative group of end users. The software is used in a natural setting with the developer “looking over the shoulder” of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a “live” application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

A variation on beta testing, called *customer acceptance testing*, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

System Testing

Software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system-testing problem is “finger pointing.” This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as “evidence” if finger pointing does occur, and (4) participate in planning

and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests that are worthwhile for software-based systems.

Recovery Testing

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing 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 (performed by the system itself), re-initialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [1984]: “The system’s security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack.”

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

Stress Testing

Earlier software testing steps resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: “How high can we crank this up before it fails?”

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called *sensitivity testing*. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of

software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

Deployment Testing

In many cases, software must execute on a variety of platforms and under more than one operating system environment. Deployment testing, sometimes called ***configuration testing***, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

As an example, consider the Internet-accessible version of *SafeHome* software that would allow a customer to monitor the security system from remote locations. The *SafeHome* WebApp must be tested using all Web browsers that are likely to be encountered. A more thorough deployment test might encompass combinations of Web browsers with various operating systems (e.g., Linux, Mac OS, Windows).

Because security is a major issue, a complete set of security tests would be integrated with the deployment test.

Software Maintenance

Software maintenance is a part of the Software Development Life Cycle. Its primary goal is to modify and update software application after delivery to correct errors and to improve performance. Software is a model of the real world. When the real world changes, the software require alteration wherever possible.

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.

Why software maintenance?

- Correct errors
- Change in user requirement with time
- Changing hardware/software requirements
- To improve system efficiency
- To optimize the code to run faster
- To modify the components
- To reduce any unwanted side effects.

Thus the maintenance is required to ensure that the system continues to satisfy user requirements.

Types of Software Maintenance

Corrective maintenance. This refer to modifications initiated by defects in the software. Corrective maintenance aims to correct any remaining errors regardless of where they may cause specifications, design, coding, testing, and documentation, etc.

Adaptive maintenance. It includes modifying the software to match changes in the ever changing environment.

Preventive Maintenance. It is the process by which we prevent our system from being obsolete. It involves the concept of reengineering & reverse engineering in which an old system with old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

Perfective maintenance. It means improving processing efficiency or performance, or restructuring the software to improve changeability. This may include enhancement of existing system functionality, improvement in computational efficiency etc.

Characteristics of Software Maintenance

Structured and Unstructured Maintenance. Unstructured maintenance happens when there is no well-defined methodology. For example, when there is no good documentation, when there is no information about the tests performed etc. Structure maintenance may be conducted only if the steps of the software engineering methodology, the waterfall cycle, are properly followed during the project development. The existence of a proper software configuration does not guarantee problem-free maintenance, but at least guarantees reduction of wasted effort and increasing the quality of change.

Causes of Software Maintenance Problems

Lack of Traceability. Codes are rarely traceable to the requirements and design specifications. It makes it very difficult for a programmer to detect and correct a critical defect affecting customer operations. Like a detective, the programmer pores over the program looking for clues. Life Cycle documents are not always produced even as part of a development project.

Lack of Code Comments. Most of the software system codes lack adequate comments. Lesser comments may not be helpful in certain situations.

Obsolete Legacy Systems. In most of the countries worldwide, the legacy system that provides the backbone of the nation's critical industries, e.g., telecommunications, medical, transportation utility services, were not designed with maintenance in mind. They were not expected to last for a quarter of a century or more!

As a consequence, the code supporting these systems is devoid of traceability to the requirements, compliance to design and programming standards and often includes dead, extra and uncommented code, which all make the maintenance task next to the impossible.

Challenges in Software Maintenance

Maintaining software is though considered essential these days, it is not a simple procedure and entails extreme efforts. The process requires knowledgeable experts who are well versed in latest software engineering trends and can perform suitable programming and testing. Furthermore, the programmers can face several challenges while executing software maintenance which can make the process time consuming and costly. Some of the challenges/problems encountered while performing software maintenance are:

Finding the person or developer who constructed the program can be difficult and time consuming.

Changes are made by an individual who is unable to understand the program clearly.

The systems are not maintained by the original authors, which can result in confusion and misinterpretation of changes executed in the program.

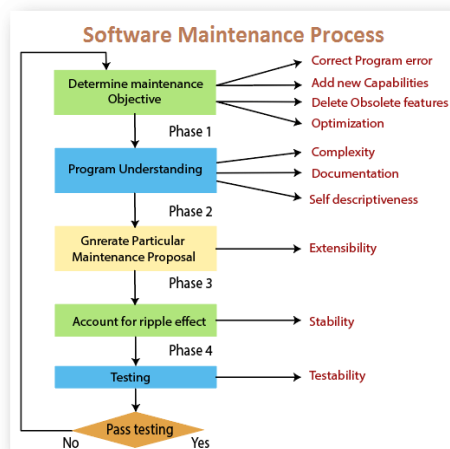
Information gap between user and the developer can also become a huge challenge in software maintenance.

The biggest challenge in software maintenance is when systems are not designed for changes.

Potential Solutions to Maintenance Problems:

- Budget and effort reallocation.
- Complete replacement of the system.
- Maintenance of existing system.

Software Maintenance Process



Program Understanding. The first step consists of analyzing the program to understand.

Generating a particular maintenance problem. The second phase consists of creating a particular maintenance proposal to accomplish the implementation of the maintenance goals.

Ripple Effect. The third step consists of accounting for all of the ripple effects as a consequence of program modifications.

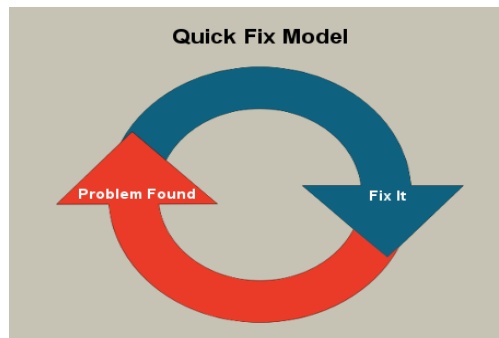
Modified Program Testing. The fourth step consists of testing the modified program to ensure that the revised application has at least the same reliability level as prior.

Maintainability. Each of these four steps and their associated software quality attributes is critical to the maintenance process. All of these methods must be combined to form maintainability.

Maintenance Models

1. Quick-fix Model

This is an adhoc approach used for maintaining the software system. The objective of this model is to identify the problem and then fix it as quickly as possible. The advantage is that it performs its work quickly and at a low cost. This model is an approach to modify the software code with little consideration for its impact on the overall structure of the software system.

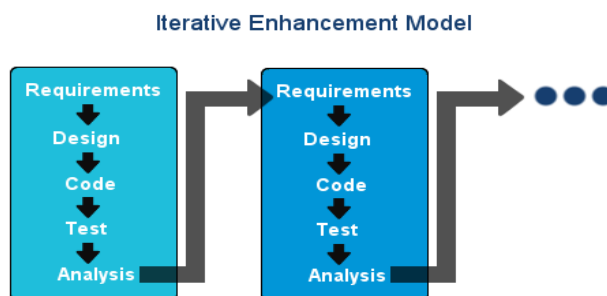


Sometimes, users do not wait for long time. Rather, they require the modified software system to be delivered to them in the least possible time. As a result, the software maintenance team needs to use a quick-fix model to avoid the time consuming process of SMLC.

This model is beneficial when a single user is using the software system. As the user has proper knowledge of the software system, it becomes easier to maintain the software system without having need to manage the detailed documentation. This model is also advantageous in situations when the software system is to be maintained with certain deadlines and limited resources. However, this model is not suitable to fix errors for a longer period.

2. Iterative Enhancement Model

Iterative enhancement model considers the changes made to the system are iterative in nature. This model incorporates changes in the software based on the analysis of the existing system. It assumes complete documentation of the software is available in the beginning. Moreover, it attempts to control complexity and tries to maintain good design.



Iterative Enhancement Model comprises three stages:

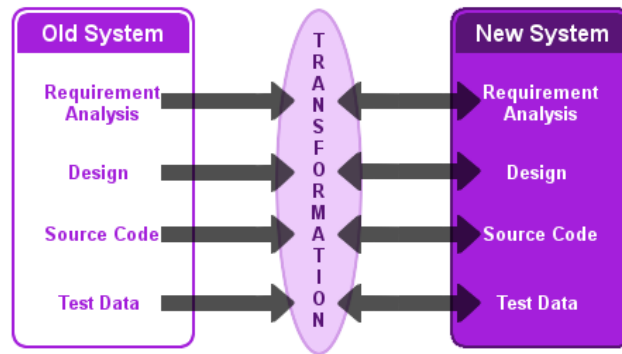
- Analysis of software system.
- Classification of requested modifications.
- Implementation of requested modifications.

In the analysis stage, the requirements are analyzed to begin the software maintenance process. *After analysis, the requested modifications are classified according to the complexity, technical issues, and identification of modules that will be affected.* At the end, the software is modified to implement the modification request. At each stage, the documentation is updated to accommodate changes of requirements analysis, design, coding, and testing phases.

3. Re-use Oriented Model

The reuse-oriented model assumes that the existing program components can be reused to perform maintenance. The parts of the old/existing system that are appropriate for reuse are identified and understood, in Reuse Oriented Model. These parts are then go through modification and enhancement, which are done on the basis of the specified new requirements. The final step of this model is the integration of modified parts into the new system.

Reuse Oriented Model

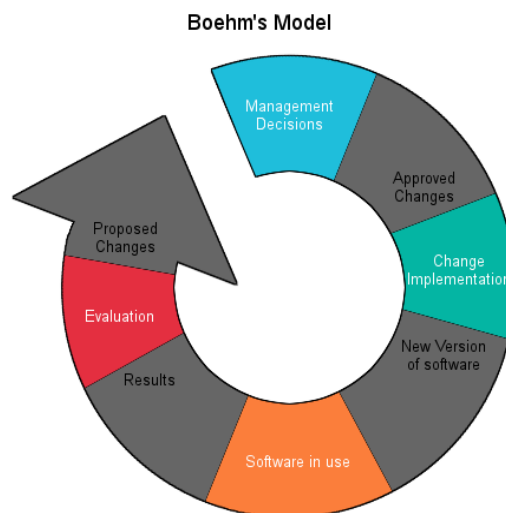


It consists of the following steps:

- Identifying the components of the old system which can be reused.
- Understanding these components.
- Modifying the old system components so that they can be used in the new system.
- Integrating the modified components into the new system.

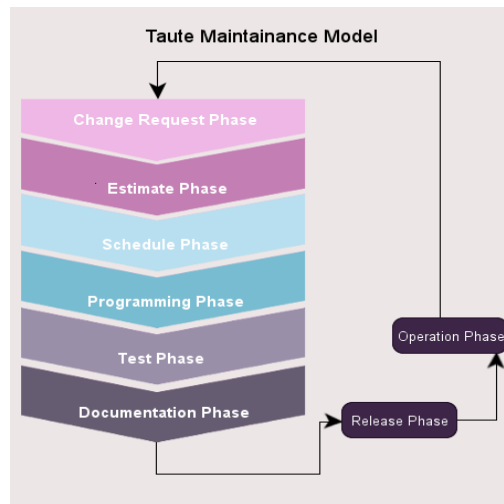
4. Boehm's Model

Boehm proposed a model for the maintenance process based upon the economic models and principles. It represents the maintenance process as a closed loop cycle, wherein changes are suggested and approved first and then are executed.



5. Taute Maintenance Model

Taute's model is a typical maintenance model that consists of eight phases in cycle fashion. The process of maintenance begins by requesting the change and ends with its operation. The phases are shown in Figure below:



Remarks:

Software maintenance has recently gained its importance in the software development process. It is one such practice which is immensely beneficial. Executed through various techniques and software maintenance models, it performs functions that fixes new or existing bugs and defects in the system and provides improved performance with regular upgrades.

Through software maintenance, software systems can adapt to the changing technical environment and latest market trends. It also helps in predicting cash flow and controlling software expenditure. Hence, by adopting software maintenance developers can provide clients services that are up-to-date with the latest trends and is extremely beneficial.

Software Configuration Management

When we develop software, the product (software) undergoes many changes in their maintenance phase; we need to handle these changes effectively.

The elements that comprise all information produced as a part of the software process are collectively called a software configuration.

As software development progresses, the number of Software Configuration elements/items (SCI's) grow rapidly. These are handled and controlled by SCM. This is where we require software configuration management. A configuration of the product refers not only to the product's constituent but also to a particular version of the component.

The process of controlling software development and maintenance is called configuration management. The configuration management is different in development and maintenance phases of life cycle due to different environments.

SCM activities are broadly categorised as:

- Identification of the components and changes.
- Monitoring and controlling of the way by which changes are made.
- Auditing and reporting on the changes made.
- Status accounting recording and documenting all the activities that have taken place.

The following documents are required for these activities:

- Project plan.
- Software requirements specification document.
- Software design description document.

Source code listing.
Test plans/procedures/test cases
User manuals.

Configuration Management (CM) is a technique of identifying, organizing and controlling modification to software being built by a programming team.

The objective is to maximize productivity by minimizing mistakes (errors).

Why Configuration Management?

Multiple people are working on software which is consistently updated. It may be a method where multiple version, branches, authors are involved in a software project, and the team is geographically distributed and works concurrently. It changes in user requirements, and policy, budget, schedules need to be accommodated.

Importance of SCM

It is practical in controlling and managing the access to various SCIs e.g., by preventing two members of a team for checking out the same component for modification at the same time.

It provides the tool to ensure that changes are being properly implemented.

It has the capability of describing and storing the various constituent of software.

SCM is used in keeping a system in a consistent state by automatically producing derived version upon modification of the same component.

QUICK RECAP!

Software is released to end users, and within days, bug reports filter back to the software engineering organization. Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment. And within months, another corporate group who wanted nothing to do with the software when it was released now recognizes that it may provide them with unexpected benefit. They'll need a few enhancements to make it work in their world.

The challenge of software maintenance has begun. You're faced with a growing queue of bug fixes, adaptation requests, and outright enhancements that must be planned, scheduled, and ultimately accomplished. Before long, the queue has grown long and the work it implies threatens to overwhelm the available resources. As time passes, your organization finds that it's spending more money and time maintaining existing programs than it is engineering new applications. In fact, it's not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance.

You may ask why so much maintenance is required and why so much effort is expended. Osborne and Chikofsky [1990] provide a partial answer:

Much of the software we depend on today is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time [and most were not], they were created when program size and storage space were principle concerns. They were then migrated to new platforms, adjusted for changes in machine and operating system technology and enhanced to meet new user needs—all without enough regard to overall architecture. The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running . . .

Another reason for the software maintenance problem is the mobility of software people. It is likely that the software team (or person) that did the original work is no longer around. Worse, other generations of software people have modified the system and moved on. And today, there may be no one left who has any direct knowledge of the legacy system.

The ubiquitous nature of change underlies all software work. Change is inevitable when computer based systems are built; therefore, you must develop mechanisms for evaluating, controlling, and making modifications.

The importance of understanding the problem (analysis) and developing a well-structured solution (design). The mechanics of these software engineering actions, and the techniques required to be sure you've done them correctly. Both analysis and design lead to an important software characteristic that we call maintainability.

In essence, *maintainability* is a qualitative indication of the ease with which existing software can be corrected, adapted, or enhanced. Much of what software engineering is about is building systems that exhibit high maintainability. But what is maintainability? Maintainable software exhibits effective modularity. It makes use of design patterns that allow ease of understanding. It has been constructed using well-defined coding standards and conventions, leading to source code that is self-documenting and understandable. It has undergone a variety of quality assurance techniques that have uncovered potential maintenance problems before the software is released. It has been created by software engineers who recognize that they may not be around when changes must be made. Therefore, the design and implementation of the software must "assist" the person who is making the change.

Software Supportability. In order to effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition, the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life cycle of the software. A reasonable definition of software supportability is

... the capability of supporting a software system over its whole product life. This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, manpower, or any other resource required to maintain the software operational and capable of satisfying its function.

Reengineering

Michael Hammer [1990]:

It is time to stop paving the cow paths. Instead of embedding outdated processes in silicon and software, we should obliterate them and start over. We should "reengineer" our businesses: use the power of modern information technology to radically redesign our business processes in order to achieve dramatic improvements in their performance. Every company operates according to a great many unarticulated rules. . . . Reengineering strives to break away from the old rules about how we organize and conduct our business.

Oil prices was 0000!!!!

By the end of the first decade of the twenty-first century, the hype associated with reengineering waned, but the process itself continues in companies large and small. The nexus between business reengineering and software engineering lies in a "system view."

Software is often the realization of the business rules that Hammer discusses. Today, major companies have tens of thousands of computer programs that support the "old business rules." As managers work to modify the rules to achieve greater effectiveness and competitiveness, software must keep pace. In some cases, this means the creation of major new computer-based systems. But in many others, it means the modification or rebuilding of existing applications.

Software Reengineering

When we need to update the software to keep it to the current market, without affecting its functionality, it is called software reengineering. It is a thorough process where the design of software is changed and programs are re-written.

Existing software ...

...Reverse engineering...

...Re-structuring...

...Forward engineering...

...Re-engineered software

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of the software becomes a headache. Even if software grows old with time, its functionality does not.

For instance, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of the software need more maintenance than others and they also need re-engineering.

Re-engineering process

Decide what to re-engineer. Is it the whole software or a part of it?

Perform reverse engineering in order to obtain specifications of existing software.

Restructure program if required. For example, changing function oriented programs into object oriented programs.

Re-structure data as required.

Apply forward engineering concepts in order to get re-engineered software.

Important terms used in software re-engineering

Reverse engineering. It is a process to achieve system specification by thoroughly analysing, understanding the existing system. This process can be seen as reverse SDLC model, i.e., we try to get higher abstraction level by analysing lower abstraction levels.

Program specification from reverse engineering.....

...Forward engineering....

Re-engineered software

An existing system is previously implemented design about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.

Program restructuring. It is a process to restructure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code restructuring and data restructuring or both.

Restructuring does not affect the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can removed via re-structuring.

Forward engineering. Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.

An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (due to the press of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software reengineering has been spawned by software maintenance problems that have been building for more than four decades.

A Software Reengineering Process Model

Reengineering takes time, it costs significant amounts of money, and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software reengineering.

A workable strategy is encompassed in a reengineering process model. I'll discuss the model later in this section, but first, some basic principles.

Reengineering is a rebuilding activity. To better understand it, consider an analogous activity: the rebuilding of a house. Consider the following situation. You've purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?

Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.

Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).

Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.

If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.

If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

Although these principles focus on the rebuilding of a house, they apply equally well to the reengineering of computer-based systems and applications.

To implement these principles, you can use a software reengineering process model that defines six activities, shown in Figure 6.7. In some cases, these activities occur in a linear sequence, but this is not always the case. For example, it may be that reverse engineering (understanding the internal workings of a program) may have to occur before document restructuring can commence.

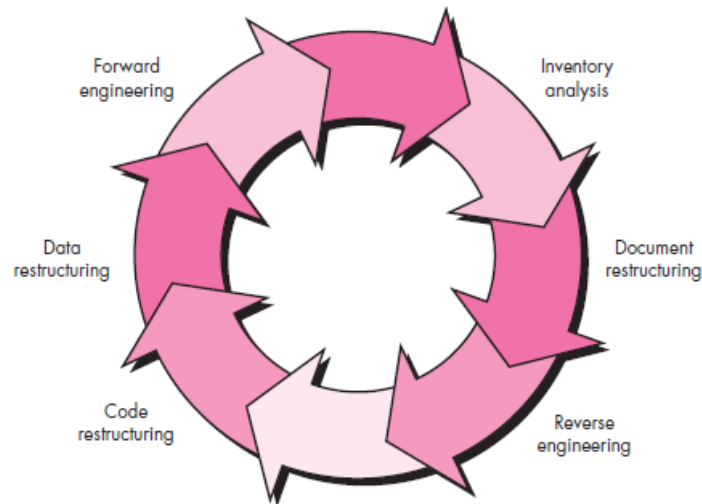


Figure 6.7. A software reengineering process model

Software Reengineering Activities

The reengineering paradigm shown in Figure 6.7 is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited. For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis. Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability and supportability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

It is important to note that the inventory should be revisited on a regular cycle. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.

Document restructuring. Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options?

Creating documentation is far too time consuming. If the system works, you may choose to live with what you have. In some cases, this is the correct approach. It is not possible to re-create documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!

Documentation must be updated, but your organization has limited resources. You'll use a "document when touched" approach. It may not be necessary to fully re-document an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.

The system is business critical and must be fully re-documented. Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Each of these options is viable. Your software organization must choose the one that is most appropriate for each case.

Reverse engineering. The term reverse engineering has its origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets." These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained. But these documents are proprietary and unavailable to the company doing the reverse engineering. In essence, successful reverse engineering derives one or more design and manufacturing specifications for a product by examining actual specimens of the product.

Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. *Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.*

Code restructuring. The most common type of reengineering (actually, the use of the term reengineering is questionable in this case) is code restructuring. Some legacy systems have a relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured (this can be done automatically) or even rewritten in a more modern programming language. The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring. A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, information architecture has more to do with the long-term viability of a program than the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.

Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward engineering. In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software re-implements the function of the existing system and also adds new functions and/or improves overall performance.

Reverse Engineering

Reverse engineering conjures an image of the "magic slot." You feed a haphazardly designed, undocumented source file into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist. Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable.

The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations (a low-level abstraction), program and data structure information (a somewhat higher level of

abstraction), object models, data and/or control flow models (a relatively high level of abstraction), and entity relationship models (a high level of abstraction). As the abstraction level increases, you are provided with information that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple architectural design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.

Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. Interactivity refers to the degree to which the human is “integrated” with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.

The reverse engineering process is represented in Figure 6.8. Before reverse engineering activities can commence, unstructured (“dirty”) source code is restructured so that it contains only the structured programming constructs. This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.

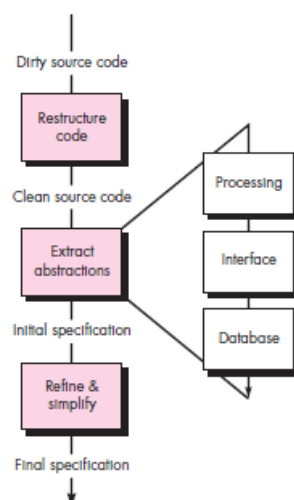


Figure 6.8. The reverse engineering process.

The core of reverse engineering is an activity called *extract abstractions*. You must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

Reverse Engineering to Understand Data

Reverse engineering of data occurs at different levels of abstraction and is often the first reengineering task. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems). Reverse engineering of the current global data structures sets the stage for the introduction of a new system wide database.

Internal data structures. Reverse engineering techniques for internal program data focus on the definition of classes of objects. This is accomplished by examining the program code with the intent of grouping related program variables. In many cases, the data organization within the code identifies abstract data types. For example, record structures, files, lists, and other data structures often provide an initial indicator of classes.

Database structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

The following steps [Pre94] may be used to define the existing data model as a precursor to reengineering a new database model: (1) build an initial object model, (2) determine candidate keys (the attributes are examined to determine whether they are used to point to another record or table; those that serve as pointers become candidate keys), (3) refine the tentative classes, (4) define generalizations, and (5) discover associations using techniques that are analogous to the CRC approach. Once information defined in the preceding steps is known, a series of transformations [Premerlani, 1994] can be applied to map the old database structure into a new database structure.

Reverse Engineering to Understand Processing

Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application system must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within the system. Each of the programs that make up the application system represents a functional abstraction at a high level of detail. A block diagram, representing the interaction between these functional abstractions, is created. Each component performs some subfunction and represents a defined procedural abstraction. A processing narrative for each component is developed. In some situations, system, program, and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.

Things become more complex when the code inside a component is considered. You should look for sections of code that represent generic procedural patterns. In almost every component, a section of code prepares data for processing (within the module), a different section of code does the processing, and another section of code prepares the results of processing for export from the component. Within each of these sections, you can encounter smaller patterns; for example, data validation and bounds checking often occur within the section of code that prepares data for processing.

For large systems, reverse engineering is generally accomplished using a semi-automated approach. Automated tools can be used to help you understand the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

Reverse Engineering User Interfaces

Sophisticated GUIs have become de rigueur for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

To fully understand an existing user interface, the structure and behavior of the interface must be specified. Merlo and his colleagues [1993] suggest three basic questions that must be answered as reverse engineering of the UI commences:

What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?

What is a compact description of the behavioral response of the system to these actions?

What is meant by a “replacement,” or more precisely, what concept of equivalence of interfaces is relevant here?

Behavioral modeling notation can provide a means for developing answers to the first two questions. Much of the information necessary to create a behavioral model can be obtained by observing the external manifestation of the existing interface. But additional information necessary to create the behavioural model must be extracted from the code.

It is important to note that a replacement GUI may not mirror the old interface exactly (in fact, it may be radically different). It is often worthwhile to develop a new interaction metaphor. For example, an old UI requests that a user provide a scale factor (ranging from 1 to 10) to shrink or magnify a graphical image. A reengineered GUI might use a slide-bar and mouse to accomplish the same function.