CODING AND TESTING

Coding

Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

• A coding standard gives a uniform appearance to the codes written by different engineers.

• It enhances code understanding.

• It encourages good programming practices.

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

Coding standards and guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

The following are some representative coding standards.

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.

Contents of the headers preceding codes for different modules: The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:

• Name of the module.

• Date on which the module was created.

• Author's name.

• Modification history.

• Synopsis of the module.

• Different functions supported, along with their input/output parameters.

• Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constant identifiers: A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

Error return conventions and exception handling mechanisms: The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

The following are some representative coding guidelines recommended by many software development organizations.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.

Avoid obscure side effects: The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

• Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.

• Use of variables for multiple purposes usually makes future enhancements more difficult.

The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

The length of any function should not exceed 10 source lines: A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use goto statements: Use of goto statements makes a program unstructured and makes it very difficult to understand.

Code review

Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module. These two types code review techniques are code inspection and code walk through.

Code Walk Through

Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk through meeting to read and understand code. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.

Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following.

• The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.

• Discussion should focus on discovery of errors and not on how to fix the discovered errors.

• In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code Inspection

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently

committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

• Use of uninitialized variables.

• Jumps into loops.

• Nonterminating loops.

• Incompatible assignments.

• Array indices out of bounds.

• Improper storage allocation and deallocation.

• Mismatches between actual and formal parameter in procedure calls.

• Use of incorrect logical operators or incorrect precedence among operators.

• Improper modification of loop variables.

• Comparison of equally of floating point variables, etc.

Clean room testing

Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. The software development philosophy is based on avoiding software defects by using a rigorous inspection process. The objective of this software is zero-defect software.

The name ‘clean room’ was derived from the analogy with semi-conductor fabrication units. In these units (clean rooms), defects are avoided by manufacturing in ultra-clean atmosphere. In this kind of development, inspections to check the consistency of the components with their specifications has replaced unit-testing.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.

The clean room approach to software development is based on five characteristics:

• Formal specification: The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.

• Incremental development: The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.

• Structured programming: Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification.

• Static verification: The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.

• Statistical testing of the system: The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on the operational profile which is developed in parallel with the system specification.

The main problem with this approach is that testing effort is increased as walk throughs, inspection, and verification are time-consuming.

Software documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

• Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.

• Use documents help the users in effectively using the system.

• Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.

• Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

Different types of software documents can broadly be classified into the following:

• Internal documentation

• External documentation

Internal documentation is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc. Careful experiments suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without thought. For

example, the following style of code commenting does not in any way help in understanding the code.

a = 10; /* a made 10 */

But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

Program Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

• Failure: This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

• Test case: This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

• Test suite: This is the set of all test cases with which a given software product is to be tested.

Aim of testing

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in asoftware product. Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Differentiate between verification and validation.

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification.

Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

Design of test cases

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible. Actually, if test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered. Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

If (x>y) max = x;

else max = x;

For the above code segment, the test suite, {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3);(x=5,y=1)} does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

Functional testing vs. Structural testing

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing.

On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

Testing in the large vs. testing in the small

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

Unit testing

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

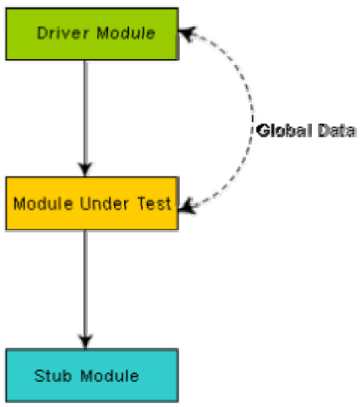
In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module:

• Theprocedures belonging to other modules that the module under test calls.

• Nonlocal data structures that the module accesses.

• A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in fig. 10.1. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.



Black box testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required. The following are the two main approaches to designing black box test cases.

• Equivalence class portioning

• Boundary value analysis

Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind

defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example#1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

The equivalence classes are the following:

• Parallel lines (m1=m2, c1≠c2)

• Intersecting lines (m1≠m2)

• Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

Boundary Value Analysis

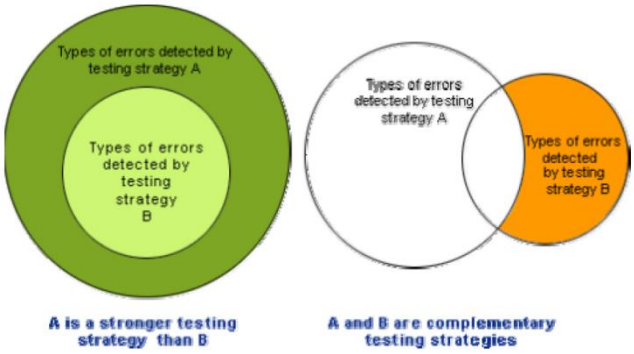
A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1,5000,5001}.

White box testing

One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two

testing strategies detect errors that are different at least with respect to some types of errors, then they are called complementary. The concepts of stronger and complementary testing are schematically illustrated in fig. 10.2.



Statement coverage

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid's GCD computation algorithm:

intcompute_gcd(x, y)

int x, y;

{

1 while (x! = y){

2 if (x>y) then

3 x= x – y;

4 else y= y – x;

5 }

6 return x;

}

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed at least once.

It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid's GCD

computation algorithm , the test cases for branch coverage can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

Condition coverage

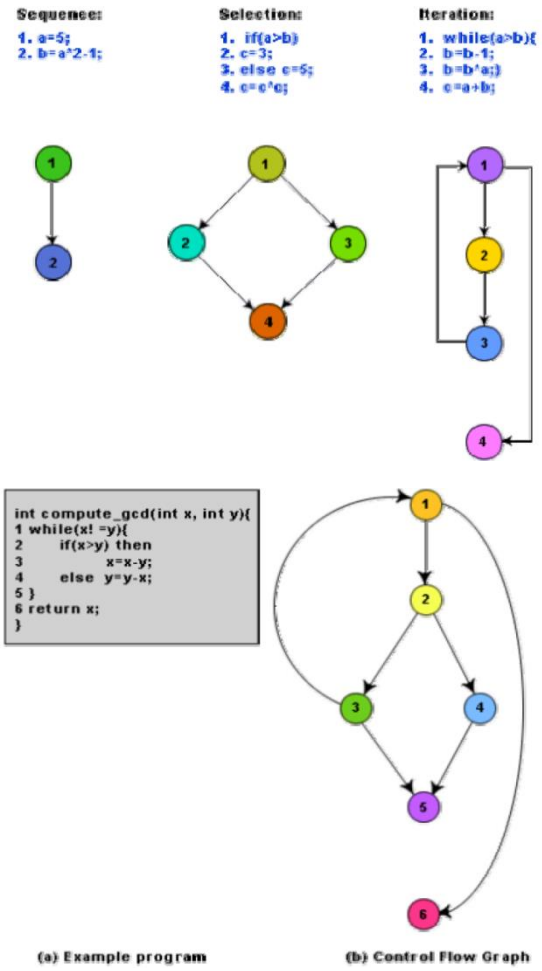
In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values. Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Path coverage

The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control Flow Graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig. 10.3). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from thesetypes of statements. Fig. 10.3 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown in fig. 10.4.



Path

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly independent path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Control flow graph

In order to understand the path coverage-based testing strategy, it is very much necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

Linearly independent path

The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

Cyclomatic complexity

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, theMcCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree.

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

V(G) = E – N + 2

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in fig. 10.4, E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = Total number of bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is notplanar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown in fig. 10.4, from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing with this method is also 2+1 = 3. This method provides a very easy way of computing the

cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1.

Data flow-based testing

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.

For a statement numbered S, let

DEF(S) = {X/statement S contains a definition of X}, and

USES(S) = {X/statement S contains a use of X}

For the statement S:a=b+c;, DEF(S) = {a}. USES(S) = {b,c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.

The definition-use chain (or DU chain) of a variable X is of form [X, S, S1], where S and S1 are statement numbers, such that X ∈ DEF(S) and X ∈ USES(S1), and the definition of X in the statement S is live at statement S1. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

Mutation testing

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant. The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc. A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically.

Need for debugging

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

Debugging approaches

The following are some of the approaches popularly adopted by programmers for debugging.

Brute Force Method:

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

Backtracking:

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

Cause Elimination Method:

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program Slicing:

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002].

Debugging guidelines

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

• Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.

• Debugging may sometimes even require full redesign of the system. In such cases, a common mistakes that novice programmers often make is attempting not to fix the error but its symptoms.

• One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

Program analysis tools

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc. We can classify these into two broad categories of program analysis tools:

• Static Analysis tools

• Dynamic Analysis tools

Static program analysis tools

Static analysis tool is also a program analysis tool. It assesses and computes various characteristics of a software product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

• Whether the coding standards have been adhered to?

• Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walk throughs and code inspections might be considered as static analysismethods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

Dynamic program analysis tools

Dynamic program analysis techniques require the program to be executed and its actual behavior recorded. A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces). The instrumented code when executed allows us to record the behavior of the software for different test cases. After the software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool caries out a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete test suite for the program. For example, the post execution dynamic analysis report might provide data on extent statement, branch and path coverage achieved.

Normally the dynamic analysis results are reported in the form of a histogram or a pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily and provides evidence that thorough testing has been done. The dynamic analysis results the extent of testing performed in white-box mode. If the testing coverage is not satisfactory more test cases can be designed and added to the test suite. Further, dynamic analysis results can help to eliminate redundant test cases from the test suite.

Integration testing

The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested. An important factor that guides the integration plan is the module dependency graph. The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart the integration plan can be developed.

Integration test approaches

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

- Big bang approach
- Top-down approach
- Bottom-up approach
- Mixed-approach

Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

Bottom-Up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data

interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

Top-Down Integration Testing

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

Phased vs. incremental testing

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

• In incremental integration testing, only one new module is added to the partial system each time.

• In phased integration, a group of related modules are added to the partial system each time.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system in the incremental testing approach since it is known that the error is caused by addition of a single module. In fact, big bang testing is a degenerate case of the phased integration testing approach.

System testing

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

• Alpha Testing. Alpha testing refers to the system testing carried out by the test team within the developing organization.

• Beta testing. Beta testing is the system testing performed by a select group of friendly customers.

• Acceptance Testing. Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the SRS document. Broadly, these tests can be classified into functionality and performance tests. The functionality tests test the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance tests test the conformance of the system with the nonfunctional requirements of the system.

Performance testing

Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document. There are several types of performance testing. Among of them nine types are discussed below. The types of performance testing to be carried out on a system depend on the different non-functional requirements of the system documented in the SRS document. All performance tests can be considered as black-box tests.

• Stress testing

• Volume testing

• Configuration testing

• Compatibility testing

• Regression testing

• Recovery testing

• Maintenance testing

• Documentation testing

• Usability testing

Stress Testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black box tests

which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multiprogrammed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours. For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

Volume Testing

It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations. For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

Configuration Testing

This is used to analyze system behavior in various hardware and software configurations specified in the requirements. Sometimes systems are built in variable configurations for different users. For instance, we might define a minimal system to serve a single user, and other extension configurations to serve additional users. The system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.

Compatibility Testing

This type of testing is required when the system interfaces with other types of systems. Compatibility aims to check whether the interface functions perform as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression Testing

This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

Recovery Testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as applicable and discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance Testing

This testing addresses the diagnostic programs, and other procedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation Testing

It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance.

Usability Testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, report formats, and other aspects relating to the user interface requirements are tested.

Error seeding

Sometimes the customer might specify the maximum number of allowable errors that may be present in the delivered system. These are often expressed in terms of maximum number of allowable errors per line of source code. Error seed can be used to estimate the number of residual errors in a system.

Error seeding, as the name implies, seeds the code with some known errors. In other words, some artificial errors are introduced into the program artificially. The number of these seeded errors detected in the course of the standard testing procedure is determined. These values in conjunction with the number of unseeded errors detected can be used to predict:

• The number of errors remaining in the product.

• The effectiveness of the testing strategy.

Let N be the total number of defects in the system and let n of these defects be found by testing.

Let S be the total number of seeded defects, and let s of these defects be found during testing.

$n/N = s/S$

or

$N = S \times n/s$

Defects still remaining after testing = N–n = n×(S – s)/s

Error seeding works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. To some extent, the different categories of errors that remain can be estimated to a first approximation by analyzing historical data of similar projects. Due to the shortcoming that the types of seeded errors should match closely with the types of errors actually existing in the code, error seeding is useful only to a moderate extent.

.

Intermediate COCOMO model:

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. So The intermediate COCOMO model uses a set of 15 cost drivers (multipliers) based on various attributes of software development. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. In general, the cost drivers can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.

Development Environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

So the effort in person month can be estimated as:

$E = ai\ (KLOC)^{bi}\ X\ EAF$

E- the effort applied in person-months.

EAF - An effort adjustment factor (EAF) typical values range from 0.9 to 1.4.

ai, bi- constants for different products

Software project	ai	bi
organic	3.2	1.05
semi-detached	3.0	1.12
embedded	2.8	1.20

Example:
Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that effort adjustment factor for this project is 1.2. Determine the effort required to develop the software product.

From the intermediate COCOMO estimation formula for organic software:

Effort= ai $(KLOC)^{bi}$ X EAF

Effort = 3.2 **x** $\mathbf{(32)^{1.05}}$ X 1.2 = 146 **PM**

Complete COCOMO model:

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some embedded.

The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

So the effort in person month can be estimated as:

Effort= $[(KLOC \times B^{0.333}) / P]^3 \times (1 / T^4)$

Where

Effort - effort in person-months or person-years.

T - Project duration in months or years

B - Special skill factor.

For small programs (KLOC = 5K – 15K), B = 0.16.

For programs greater than 70 KLOC, B = 0.39.

P - "Productivity parameter" that reflects:

- Overall process maturity and management practices
- The extent to which good software engineering practices are used
- The level of programming languages used
- The state of the software environment
- The skill and experience of the software team
- The complexity of the application

P = 2,000 for real-time embedded software

P = 10,000 for telecommunication systems,

P = 28,000 for business application systems

3.9 STAFFING LEVEL ESTIMATION

Once the effort required to complete a software project has been estimated, the staffing requirement for the project can be determined. Putnam was the first to study the problem of determining a proper staffing pattern for software projects. He extended the classical work of Norden who had earlier investigated the staffing pattern of general research and development (R&D) type of projects. In order to appreciate the uniqueness of the staffing pattern that is desirable for software projects, we must first understand both Norden's and Putnam's results.

3.9.1 Norden's Work

Norden studied the staffing patterns of several R&D projects. He found that the staffing pattern of R&D type of projects is very different from that of manufacturing or sales. In a sales outlet, the number of sales staff does not usually vary with time. For example, in a supermarket the number of sales personnel would depend on the number of sales counters and would be approximately constant over time. However, the staffing pattern of R&D type of projects needs to change over time. At the start of an R&D project, the activities of the project are planned and initial investigations are made. During this time, the manpower requirements are low. As the project progresses, the manpower requirement increases, until it reaches a peak. Thereafter, the manpower requirement gradually diminishes.

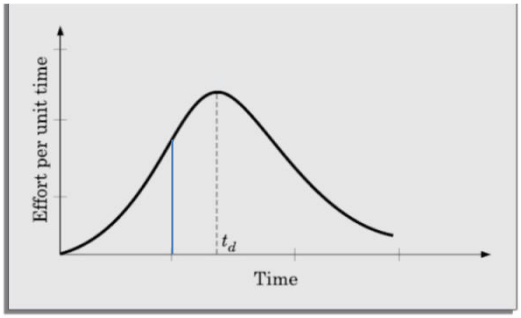


Figure 3.6: Rayleigh curve.

Norden represented the Rayleigh curve by the following equation:

$$E = \frac{K}{{t_d}^2} * t * e^{\frac{-t^2}{2{t_d}^2}}$$

where, E is the effort required at time t. E is an indication of the number of developers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and t_d is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R&D projects and were not meant to model the staffing pattern of software development projects.

3.9.2 Putnam's Work

Putnam studied the problem of staffing of software projects and found that the staffing pattern for software development projects has characteristics very similar to any other R&D projects. Only a small number of developers are needed at the beginning of a project to carry out the planning and specification tasks. As the project progresses and more detailed work is performed, the number of developers increases and reaches a peak during product testing. After implementation and unit testing, the number of project staff falls. Putnam found that the Rayleigh-Norden curve can be adapted to relate the number of delivered lines of code to the effort and the time required to develop the product. By analysing a large number of defence projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

where the different terms are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration and testing. Therefore, t_d can be approximately considered as the time required to develop the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of C_k=2 for poor development environment (no methodology, poor documentation, and review, etc.), C_k=8 for good software development environment (software engineering principles are adhered to), C_k=11 for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from historical data of the organisation developing it.

For efficient resource utilisation as well as project completion over optimal duration, starting from a small number of developers, there should be a staff build-up and after a peak size has been achieved, staff reduction is required. However, the staff build-up should not be carried out in large instalments.

The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve. It should be clear that a constant level of manpower throughout the project duration would lead to wastage of effort and as a result would increase both the time and effort required to develop the product. If a constant number of developers are used over all the phases of a project, some phases would be overstaffed and the other phases would be understaffed causing inefficient use of manpower, leading to schedule slippage and increase in cost. If we examine

the Rayleigh curve, we can see that approximately 40 per cent of the area under the Rayleigh curve is to the left of t_d and 60 per cent area is to the right of t_d . This has been verified mathematically by integrating the expression provided by Putnam. This implies that the effort required to develop the product to its maintenance effort is approximately in 40:60 ratio.

Effect of schedule change on cost according to Putnom method

Putnam's method (Eq. 3.2) can be used to study the effect of changing the duration of a project from that computed by the COCOMO model. By using the Putnam's expression Eq. (3.2):

$$K = \frac{L^3}{(C_k^3 t_d^4)}$$

or,

$$K = \frac{C}{{t_d}^4}$$

For the same product size, $C = \frac{L^3}{C_k^3}$ is a constant.

Or,

$$\frac{K_1}{K_2} = \frac{t_{d_2}^4}{t_{d_1}^4}$$

From this expression, it can easily be observed that when the schedule of a project is compressed, the required effort increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in substantial penalty on human effort. For example, if the estimated development time using COCOMO formulas is 1 year, then in order to develop the product in 6 months, the total effort required (and hence the project cost) increases 16 times.

Example 3.7 The nominal effort and duration of a project have been estimated to be 1000PM and 15 months. The project cost has been negotiated to be Rs. 200,000,000. The needs the product to be developed and delivered in 12 month time. What should be the new cost to be negotiated?

Answer: The project can be classified as a large project. Therefore, the new cost to be negotiated can be given by the Putnam's formula:

new cost = Rs.200, 000, 000 × (15/12)4 = Rs. 488,281,250.

3.9.3 Jensen's Model

Jensen model [Jensen 84] is very similar to Putnam model. However, it attempts to soften the effect of schedule compression on effort to make it

$$L = C_{te} t_d K^{1/2}$$

$$\frac{K_1}{K_2} = \frac{t_{d_2}^2}{t_{d_1}^2}$$

where, C_{te} is the effective technology constant, t_d is the time to develop the software, and K is the effort needed to develop the software. In contrast to the Putnam's model, Jensen's model considers the increase in effort (and cost) requirement to be proportional to the square of the degree of compression.

Project scheduling

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different engineers. The work breakdown structure formalism helps the manager to breakdown the tasks systematically.

After the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity

B. In general, the task dependencies define a partial ordering among tasks, i.e. each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities are represented in the form of an activity network.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT chart representation is developed. The PERT chart representation is suitable for program monitoring and control. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

Work breakdown structure

Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem. The root of the tree is labeled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. Fig. 11.7 represents the WBS of an MIS (Management Information System) software.

While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

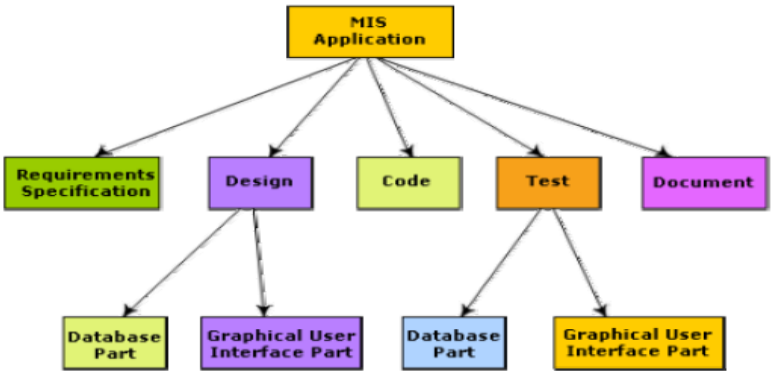


Fig. 11.7: Work breakdown structure of an MIS problem

Activity networks and critical path method

WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies (as shown in fig. 11.8). Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however is not a good idea, because software engineers often resent such unilateral decisions. A possible alternative is to let engineer himself estimate the time for an activity he can assigned to. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the engineers to do a better and faster job. However, careful experiments have shown that unrealistically aggressive schedules not only cause engineers to compromise on intangible quality aspects, but also are a cause for schedule delays. A good way to achieve accurately in estimation of the task durations without creating undue schedule pressures is to have people set their own schedules.

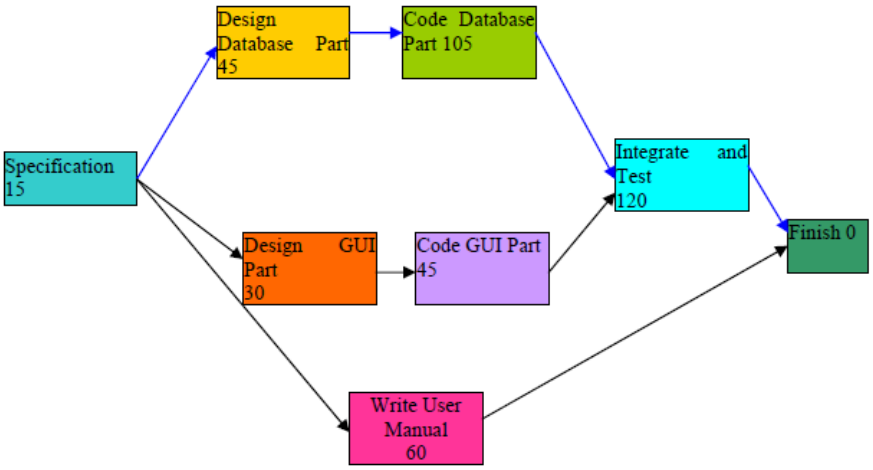


Fig. 11.8: Activity network representation of the MIS problem

Critical Path Method (CPM)

From the activity network representation following analysis can be made. The minimum time (MT) to complete the project is the maximum of all paths from start to finish. The earliest start (ES) time of a task is the maximum of all paths from the start to the task. The latest start (LS) time is the difference between MT and the maximum of all paths from this task to the finish. The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task. The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT. The slack time (ST) is LS – ES and equivalently can be written as LF – EF. The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the "flexibility" in starting and completion of tasks. A critical task is one with a zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path. These parameters for different tasks for the MIS problem are shown in the following table.

Task	ES	EF	LS	LF	ST(LF-EF)
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code database	60	165	60	165	0
Code GUI part	45	90	120	165	75
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

MT=285

The critical paths are all the paths whose duration equals MT. The critical path in fig. 11.8 is shown with a blue arrow.

Gantt chart

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of fig. 11.8 is shown in the fig. 11.9.

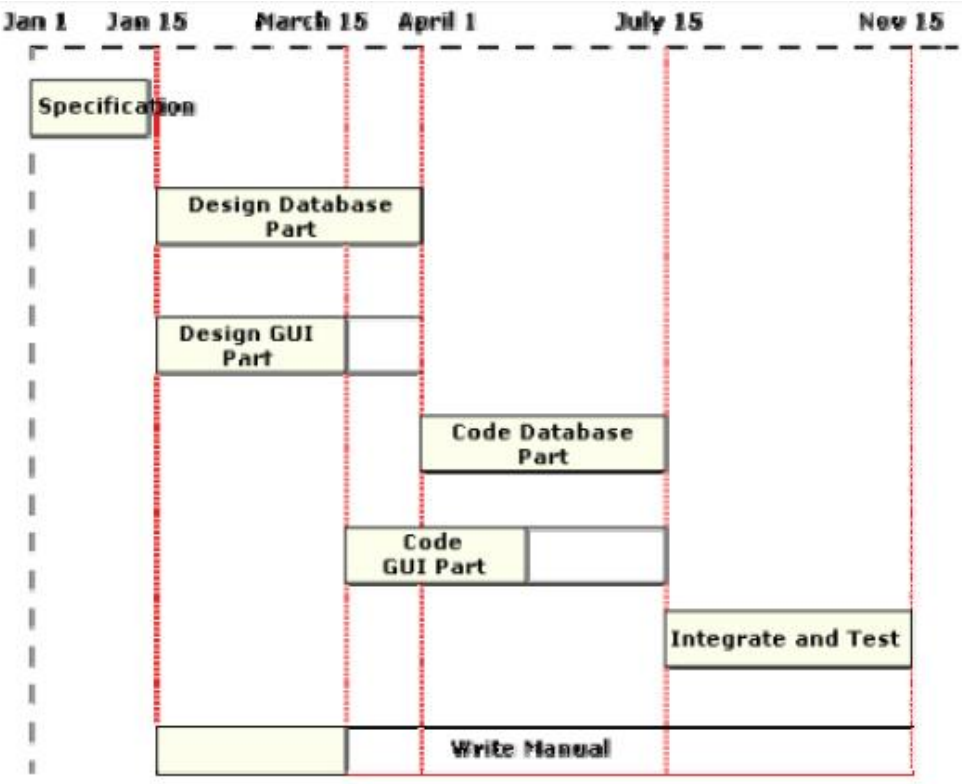
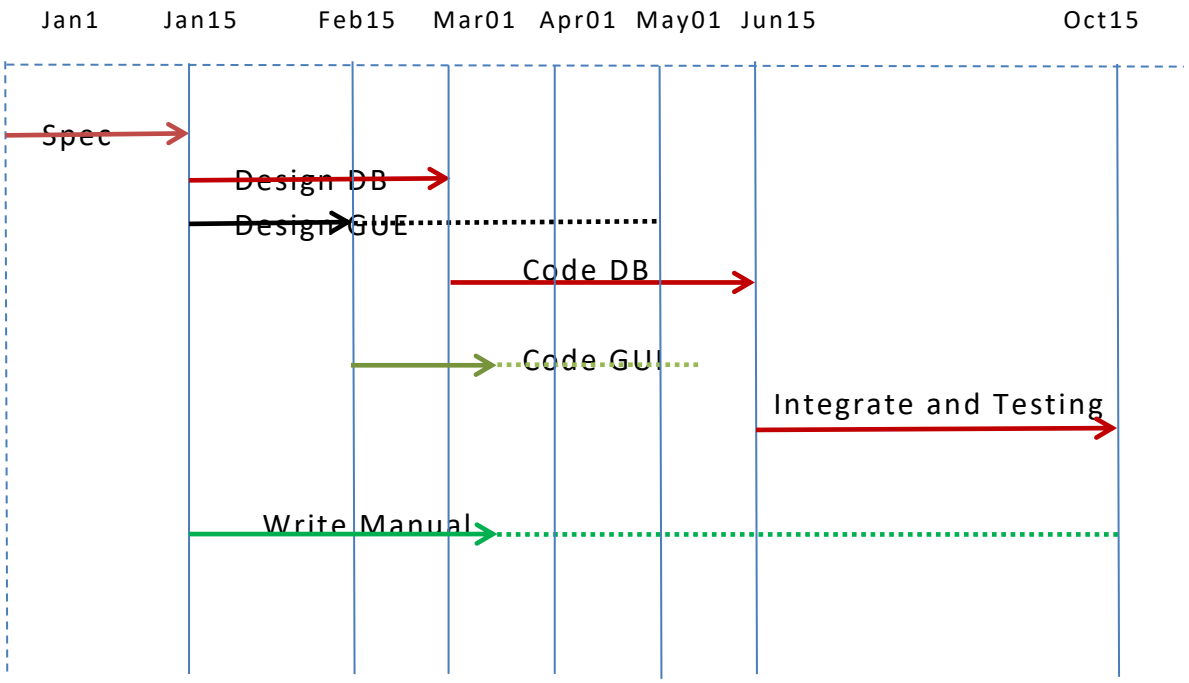


Fig. 11.9: Gantt chart representation of the MIS problem



PERT chart

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of fig. 11.8 is shown in fig. 11.10. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited.

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.

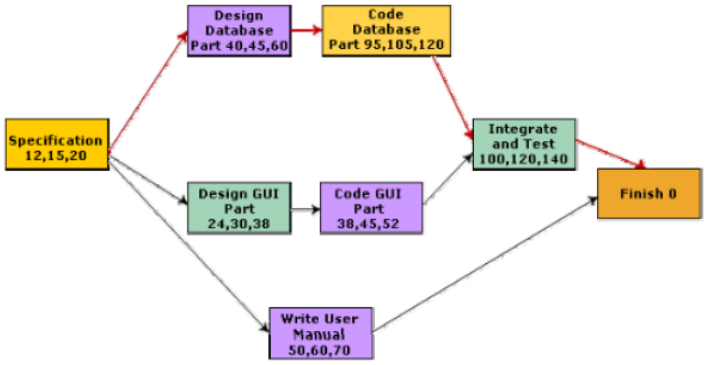


Fig. 11.10: PERT chart representation of the MIS problem

ORGANISATION AND TEAM STRUCTURES

Usually every software development organisation handles several projects at any time. Software organisations assign different teams of developers to handle different software projects. With regard to staff organisation, there are two important issues—How is the organisation as a whole structured? And, how are the individual project teams structured? There are a few standard ways in which software organisations and teams can be structured. We discuss these in the following subsection.

Organisation Structure

Essentially there are three broad ways in which a software development organisation can be structured—functional format, project format, and matrix format. We discuss these three formats in the following subsection

Functional format

In the functional format, the development staff are divided based on the specific functional group to which they belong to. This format has schematically been shown in Figure 3.13(a).

The different projects borrow developers from various functional groups for specific phases of the project and return them to the functional group upon the completion of the phase. As a result, different teams of programmers from different functional groups perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the product evolves. Therefore, the functional format requires considerable communication among the different teams and development of

good quality documents because the work of one team must be clearly understood by the subsequent teams working on the project. The functional organisation therefore mandates good quality documentation to be produced after every activity.

Project format

In the project format, the development staff are divided based on the project for which they work (See Figure 3.13(b)). A set of developers is assigned to every project at the start of the project, and remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. An advantage of the project format is that it provides job rotation. That is, every developer undertakes different life cycle activities in a project. However, it results in poor manpower utilisation, since the full project team is formed since the start of the project, and there is very little work for the team during the initial phases of the life cycle.

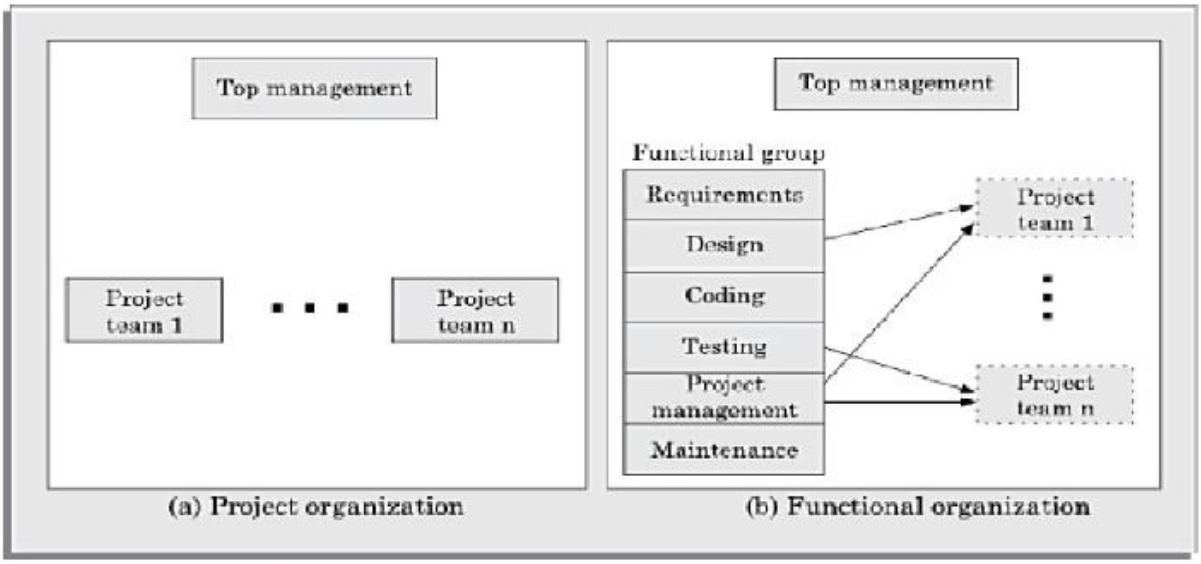


Figure 3.13: Schematic representation of the functional and project organisation.

Team Structure

Team structure addresses organisation of the individual project teams. Let us examine the possible ways in which the individual project teams are organised. In this text, we shall consider only three formal team structures—democratic, chief programmer, and the mixed control team organisations, although several other variations to these structures are possible. Projects of specific complexities and sizes often require specific team structures for efficient working.

Chief programmer team

In this team organisation, a senior engineer provides the technical leadership and is designated the chief programmer. The chief programmer partitions the task into many smaller tasks and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in Figure 3.15. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since the team members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer. That is, a project might suffer severely, if the chief programmer either leaves the organisation or becomes unavailable for some other reasons.

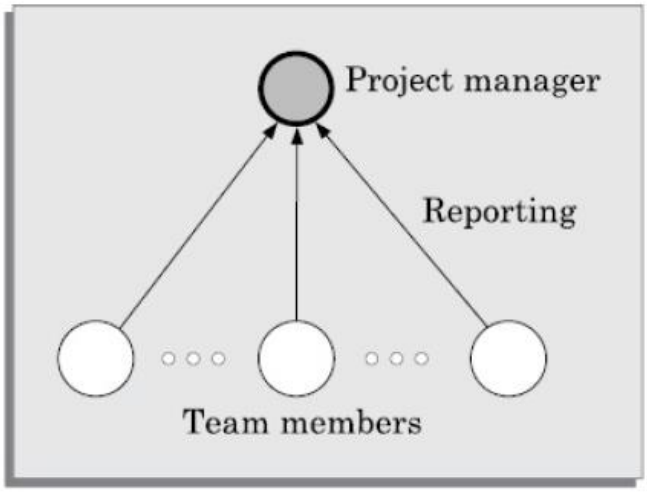


Figure 3.15: Chief programmer team structure.

Let us now try to understand the types of projects for which the chief programmer team organisation would be suitable. Suppose an organisation has successfully completed many simple MIS projects. Then, for a similar MIS project, chief programmer team structure can be adopted. The chief programmer team structure works well when the task is within the intellectual grasp of a single individual. However, even for simple and well understood problems, an organisation must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early completion outweighs other factors such as team morale, personal developments, etc.

Democratic team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy (see Figure 3.16). Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

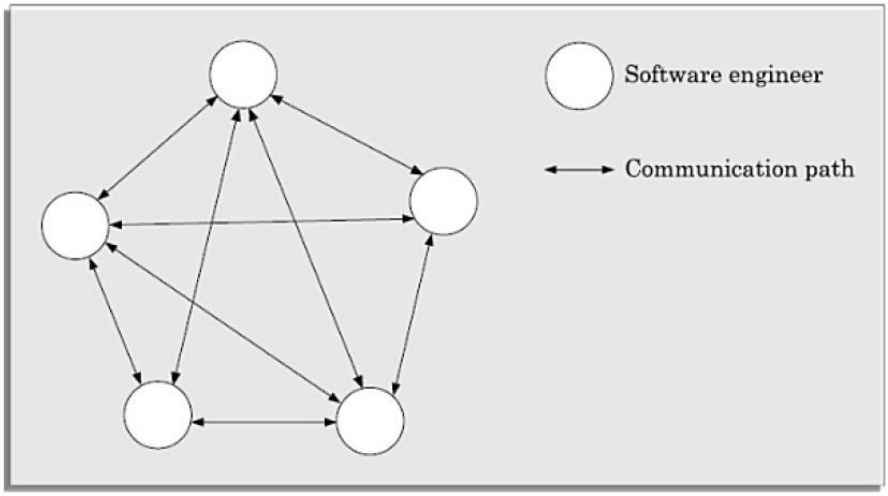


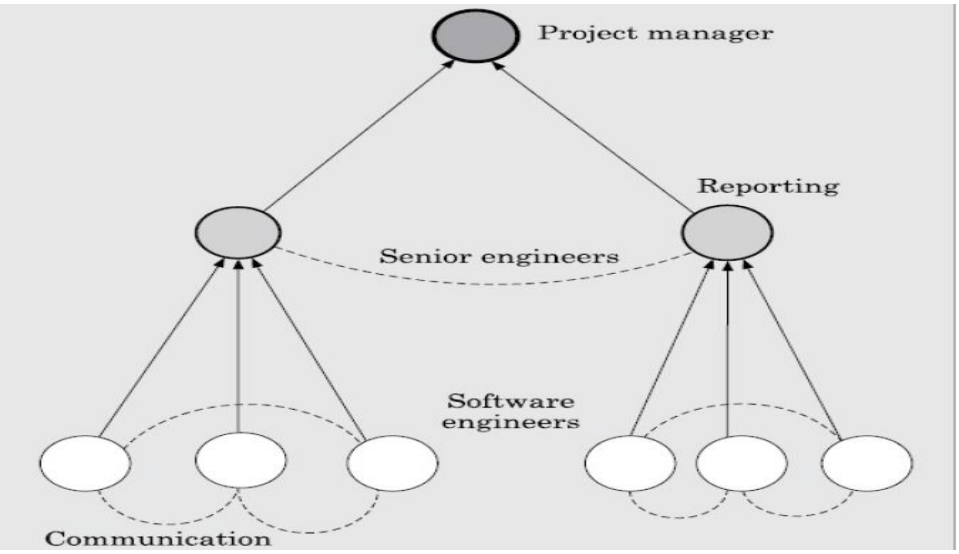
Figure 3.16: Democratic team structure.

In a democratic organisation, the team members have higher morale and job satisfaction. Consequently, it suffers from less manpower turnover. Though the democratic teams are less productive compared to the chief programmer team, the democratic team structure is appropriate for less understood problems, since a group of developers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for research-oriented projects requiring less than five or six developers. For large sized projects, a pure democratic organisation tends to become chaotic. The democratic team organisation encourages egoless programming as programmers can share and review each other's work. To appreciate the concept of egoless programming, we need to understand the concept of ego from a psychological perspective. Most of you might have heard about temperamental artists who take much pride in whatever they create. Ordinarily, the human psychology makes an individual take pride in everything he creates using original thinking. Software development requires original thinking too, although of a different type. The human psychology makes one emotionally involved with his creation and hinders him from objective examination of his creations. Just like temperamental artists, programmers find it extremely difficult to locate bugs in their own programs or flaws in their own design. Therefore, the best way to find problems in a design or code is to have someone review it.

Often, having to explain one's program to someone else gives a person enough objectivity to find out what might have gone wrong. This observation is the basic idea behind code walk throughs to be discussed in Chapter 10. An application of this, is to encourage a democratic teams to think that the design, code, and other deliverables to belong to the entire group. This is called egoless programming because it tries to avoid having programmers invest much ego in the development activity they do in a democratic set up. However, a democratic team structure has one disadvantage—the team members may waste a lot time arguing about trivial points due to the lack of any authority in the team to resolve the debates.

Mixed control team organisation

The mixed control team organisation, as the name implies, draws upon the ideas from both the democratic organisation and the chief-programmer organisation. The mixed control team organisation is shown pictorially in Figure 3.17. This team organisation incorporates both hierarchical reporting and democratic set up. In Figure 3.17, the communication paths are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organisation is suitable for large team sizes. The democratic arrangement at the senior developers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organisation is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.



Requirement Analysis & Specification

3.1 Need for Requirement Analysis

Requirement analysis is a Software engineering task that bridges the gap between system level requirements engineering and software design. Requirement analysis provides software designer with a representation of system information, function, and behavior that can be translated to data, architectural, and component-level designs.

Software requirement analysis may be divided into five areas of effort:

- ✓ Problem recognition
- ✓ Evaluation and synthesis
- ✓ Modeling
- ✓ Specification
- ✓ Review

3.2 Software Requirement Specification Principle

Specification principles are:

• Separate functionality from implementation.

• Develop a behavioral model that describes functional responses to all system stimuli.

• Define the environment in which the system operates and indicate how the collection of agents will interact with it.

• Create a cognitive model rather than an implementation model

• Recognize that the specification must be extensible and tolerant of incompleteness.

• Establish the content and structure of a specification so that it can be changed easily.

3.3 SRS document

A software requirements specification (**SRS document**) describes how a software system should be developed. ... It offers high-grade definitions for the functional and non-functional specifications of the software, and can also include use cases that illustrate how a user would interact with the system upon completion.

The important parts of SRS document are:
Functional requirements of the system
Non-functional requirements of the system, and

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions {*fi*}. The functional view of the system is shown in fig. 5.1. Each function fi of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (*oi*). The user can get some meaningful piece of work done using a high-level function.

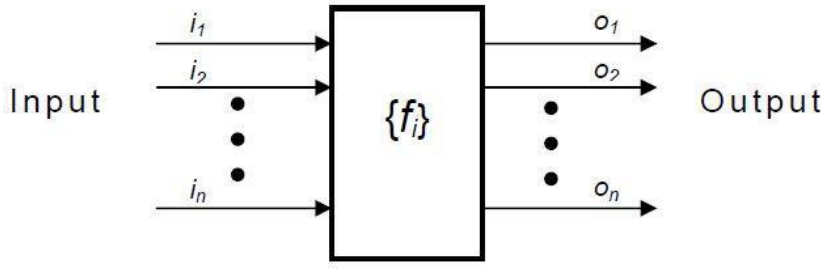


Fig. 5.1: View of a system performing a set of functions

Nonfunctional requirements:-

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

3.3 Phases of SRS Document

The requirements analysis and specification phase starts once the feasibility study phase is completed and the project is found to be financially sound and technically feasible. The goal of the requirement analysis and specification phase is to clearly understand the customer requirements and to systematically organize these requirements in a specification document.

This phase consists of two activities:

- Requirements gathering and analysis.

- Requirements specification

System analysts collect data pertaining to the product to be developed and analyze these data to conceptualize what exactly needs to be done. The analyst starts the requirements gathering and analysis activity by the collecting all information from the customer which could be used to develop the requirements of the system. The analyst then analyzes the collect information to obtain a clear and thorough understanding of the product to be developed.

Two main activities involved in the requirements gathering and analysis

phase are:

- ✓ Requirements Gathering: The activity involves interviewing the end users and customers and studying the existing documents to collect all possible information regarding the system.
- ✓ Analysis of Gathered Requirements : The main purpose of this activity is to clearly understand the exact requirements of the customer.

The analyst should understand the problems:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly the data output required of the system?
- What are the complexities that might arise while solving the problem?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems.

There are three main types of problems in the requirement that analyst

needs to identify and resolve:

- ✓ Anomaly
- ✓ Inconsistency
- ✓ Incompleteness

Anomaly: An anomaly is an ambiguity in the requirement. When a requirement is anomalous, several interpretation of the requirement are possible.

Example: In a process control application, a requirement expressed by one user is that when the temperature becomes high, the heater should be switched off. (Words such as high, low, good, bad etc, are ambiguous without proper quantification). If the threshold above which the temperature can be considered to be high is not specified, then it can be interpreted differently by different people.

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other two-end user of the system give inconsistent description of the requirement.

Example: For the case study of the office automation, one of the clerk described that a student securing fail grades in three or more subjects should have to repeat the entire semester. Another clerk mentioned that there is no provision for any student repeat a semester. Incompleteness: An incomplete set of requirements is one in which some requirements have been overlooked.

Software Requirement Specification

After the analyst has collected all the required information regarding the software to be developed and has removed all incompleteness, inconsistencies and anomalies from the specification, analyst starts to systematically organize the requirements in the form of an SRS document.

The SRS document usually contains all the user requirements in an informal form. Different People need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs are as follows.

- Users, customers and marketing personnel The goal of this set of audience is to ensure that the system as describe in the SRS document will meet their needs.

- The software developers refer to the SRS document to make sure that they develop exactly what is required by the customer.

- Test Engineers: Their goal is to ensure that the requirements are understandable from a functionality point of view, so that they can test the software and validate its working.

- User Documentation Writers: Their goal in reading the SRS document is to ensure that they understand the document well enough to be able to write the users' manuals.

- Project Managers: They want to ensure that they can estimate the cost of the project easily by referring to be SRS document and that it contains all information required to plan the project.

- Maintenance Engineers: The SRS document helps the maintenance engineers to understand the functionalities of the system. A clear knowledge of the functionalities can help them to understand the design and code.

Contents of the SRS Document

An SRS document should clearly document:

- Functional Requirements

- Nonfunctional Requirements

- Goals of implementation

The functional requirements of the system as documented in the SRS document should clearly describe each function which the system would support along with the corresponding input and output data set.

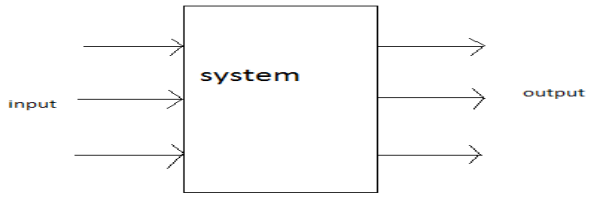


Fig. 3.1 Contents of SRS Document

The non-functional requirements also known as quality requirements. The non-functional requirements deal with the characteristics of the system that cannot be expressed as functions.

Examples of nonfunctional requirements include aspects concerning maintainability, portability and usability, accuracy of results. Non-functional requirements arise due to user requirements, budget constraints, organizational policies and soon.

The goals of implementation part of the SRS document gives some general suggestion regarding development. This section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future.

3.4 Characteristics and Organization of SRS Document

Characteristics of SRS document

Concise: The SRS document should be concise, unambiguous, consistent and complete. Irrelevant description reduced readability and also increases error possibilities.

Structured: The SRS document should be well-structured. A well-structured document is easy to understand and modify.

Block-box View: It should specify what the system should do. The SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS should specify the externally visible behavior of the system. [For this reason the SRS document is called the block-box specification of a system.]

Conceptual Integrity : The SRS document should exhibit conceptual integrity so that the reader can easily understand the contents.

Verifiable: All requirements of the system as documented in the SRS document should be verifiable if and only if there exists some finite cost effective process with which a person of machine can check that the software meets the requirement.

Modifiable : The SRS is modifiable if and only if its structure and style are such that any changes to the requirements can be made easily, completely and consistently while retaining the structure and style.

Organization of the SRS Document

Organization of the SRS document and the issues depends on the type of the product being developed.

Three basic issues of SRS documents are:

functional requirements, non functional requirements, and guidelines for system implementations.

The SRS document should be organized into:

1. Introduction

(a) Background

(b)Overall Description

(c)Environmental Characteristics

(i)Hardware

(ii)Peripherals

(iii)People

3.5 Techniques for representing complex logic

Before requirements can be analyzed, modeled or specified they must be gathered through an elicitation process.

Initiating the Process

• The most commonly used requirements elicitation technique is to conduct a meeting or interview. Customer meetings are the most commonly used technique.

• Use context free questions to find out customer's goals and benefits, identify stakeholders, gain understanding of problem, determine customer reactions to proposed solutions, and assess meeting effectiveness.

Facilitated Application Specification Techniques

• Meeting held at neutral site, attended by both software engineers and customers.

• Rules established for preparation and participation.

• Agenda suggested to cover important points and to allow for brainstorming.

• Meeting controlled by facilitator (customer, developer, or outsider).

• Goal is to identify problem, propose elements of solution, negotiate different approaches, and specify a preliminary set of solution requirements.

Quality Function Deployment (QFD)

Quality function deployment is a quality management technique that translates the needs of the customer into technical requirements for software.

Quality function deployment identifies three types of requirements:

- Normal requirements: The objectives and goals that are stated for a product or system during meetings with the customer.

- Expected requirements: These requirements are implicit to the product or system (customers assumes will be present in a professionally developed product without having to request them explicitly).

- Exciting requirements: These features that go beyond the customer's expectations and prove to be very satisfying when they are present.

Function deployment is used to determine the value of each function that is required for the system. Information deployment identifies both the data objects and events that the system must consume and produce. Task deployment examines the behavior of the system or product within the context of its environment. Value analysis used to determine the relative priority of requirements during function, information, and task deployment.

01

Saturday

Software - Software is a set of related program or set of instructions which has a specific objectives.

Basic Component of software:-
i) set of related Program
ii) operating Procedure.
iii) Documentation manual.

Defination of software Engg:-
→ This is a way of approaching to a problem which is related to a software.

→ software Engineering is defined as a technological discipline concern with systematic production and maintainance of software products, that are developed and modified within time and cost estimates.

02 Sunday

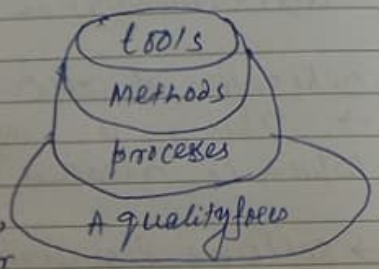
→ It is defined as the systematic approach to the development, operation, maintainance and retairement of software.

Software Engineering is a layered structure and it has 4 layers.

03 Monday

i) A quality focus.
ii) Processes
iii) Methods
iv) Tools

Diagramatically it can be represented as



i) A quality focus:-
The quality of the software is to be focus by applying some software Engineering Principles. like Analysis the Problem properly, coding or translate the objective or problem in a program by using suitable programing language, test the product at different levels or phases so that the product's Output is satisfactory. and design the problem so that it should be easier for coding and to understand.
In Other words the Engineering principles

ii) Processes: like Analysis, design, Construction, verification, and management of Technical and social entities.

ii) Processes: This is the foundation layer which include all the activities of all the phases.

04 Tuesday

it includes some Key Process Areas (KPAS)

[The Processes are the set of different activities along with constraints. All processes are combined for a project and all the projects are combined to get the product]

Key Process Areas or KPAs has Four objectives

a) Management Control over software.
b) Production of Work Product.
c) Ensuring the quality.
d) Managing the Changes properly.

a) Management Control over software:-
A software is not a single man's work, rather it is a work and co-operation of a group of persons. so to keep the co-operations among different groups. some management Control is necessary.

b) Production of Work Product:-
After Completion of each phase, the output or the work product should be produced.

c) Ensuring the quality
Since the process are set of different activities and are the building blocks of a software,

You cannot strengthen the weak by weakening the strong.

05 Wednesday

the well-defined processes can ensure the quality of a product.

iii) Managing Changes properly:
If some changes, ~~in software~~ are carried out in future, to meet the requirement of user, or to update the software, it should be done carefully so that it shouldn't affect the normal functionality.

iii) Methods:
It includes different descriptive technique and different modelling. It include certain activities like
a) Requirement Analysis and specification.
b) Design.
c) Program Construction
d) Testing support.

a) Requirement Analysis and specification:
The aim of the requirement Analysis and specification phase is to understand the exact requirements of the customer and to document them properly. It consist of two distinct activities.
1) Requirement gathering and Analysis.
2) Requirement specification.

Try making every day a day of achievement. Results count, not long hours of efforts.

Essential | Job to do | Phone No.

06 Thursday

1) Requirement Gathering and Analysis:
The goal of the requirement gathering activity is to collect all relevant information from the customer regarding the product to be developed with a view to clearly understanding the customer requirements and weeding out the incompleteness and inconsistancies in these requirements.

2) Requirement Specification:
Once the customer's requirements are gathered and analysed, these informations are organised into a SRS document. The main components of SRS documents are functional requirements which involves the identification of functions to be supported the system, and nonfunctional requirements which involve identifying the performance requirements, the required standard to be followed etc.

b) Design:
The goal of design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.

c) Program Construction:
The purpose this this phase is to translate the software design into source code. Each component of the design is implemented as a program module. Every software development organization normally formulates its own coding

Success is a result of deeds & not dreams.

Essential | Job to do | Phone No.

07 Friday

standards. A coding standard addresses issues such as the standard ways of laying out the program code, the template for laying out the functions and module headers, commenting guidlines, variable and function naming convention, maximum no. of source lines permitted in each module & so on.

d) Testing Support:-

During this phase, each module is to be tested. Each module is unit tested to determine the correct working of all individual modules. After module testing, unit testing, integration testing, system testing are carried out to debug the errors at each stage.

iv) Tools:-

It supports different Methods and processes which helps in making the software development process more efficient. Since software is becoming the costliest component in any computer installation and also it is costlier due to increased man power costs involved, different CASE tool are introduced to reduce the software development and maintainance cost and help to develop better quality products more efficiently.

Discussion is an exchange of knowledge. Argument is an exchange of ignorance.

Essential | Job to do | Phone No.

Software Life Cycle Methods

Saturday 08

A software life cycle is the series of identifiable stages that the software product undergoes during it's life time.

A Software life cycle model is a descriptive and diagrammatic representation of the software life cycle. It represents all the activities required to make a software product transit through it's life cycle phases and also capture the order in which these activities are to be undertaken.

The software life cycle model is also called software process model.

A Process covers all the activities begining from the product inception through delivery and Retirement and also address issues like reuse, documentation, testing, parallel work carried out by team members and co-ordination with customers.

A Methodology covers only a single activity or at best a few individual steps in the development.

Ex:- Testing Methodology, Design Methodology.

A life cycle model clearly defines the the entry and exit criteria for every phase so that a phase can only begen or a phase can only exit if it's phase-entry criteria or phase exit criteria are satisfied respectively.

Refusing in a kind manner is better than making promises which are not kept.

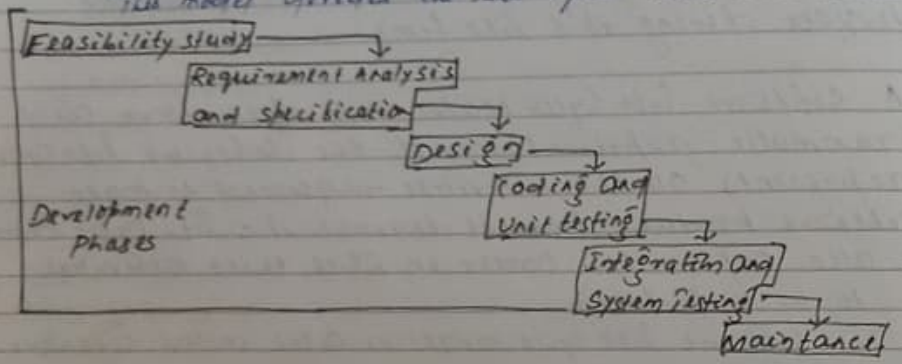
Essential | Job to do | Phone No.

10 Monday

010-355
3rd Week

i) Classical Waterfall Model:-

The model devides the life cycle into six phases.



(Classical waterflow Model)

Among all the phases the phases from feasibility study to Integration and System testing in Classical waterflow Model is called Development Phase. so in development Phase the Integration and system testing phase takes maximum efforts and if we consider the total life cycle then maintainance will take maximum effort.

1- Feasibility study:-

The objectives of this phase is to determine whether it would be financially and technically feasible to develop the product by analysing the problem and by collecting all relevant information relating to the product.

Essential | Job to do | Phone No.

011-354
3rd Week

11 Tuesday

2- Requirement Analysis and Specification:- (What to do)

The main objective of this phase is to understand the exact requirements of the customer and to document them properly. This phase is devided into two sub activity.

i) Requirement Gathering and Analysis.
ii) Requirement Specification.

i) Requirement Gathering and Analysis:- (Custom-Oriented requirement)

The main objective of this activity is to collect all the relevent information from the customer regarding the product to be developed, through interviews and discussion, with a view to clearly understanding the customer requirements and weeding out the incompleteness and inconsistancies in these requirements.

ii) Requirement specification:- (developer-Oriented Requirement)

After all the ambiguities, inconsistancies and incompleteness have been resolved and all requirements properly understood, the requirement specification activity can start where the user requirements are systematically organised in a SoftWare Requirement specification (SRS) document.

This document contains both functional and non functional requirements of the system.

The functional requirements include identification of input data, the processing required on the input data and

Essential | Job to do | Phone No.

12 Wednesday

the output data to be produced.

The non-functional requirements identify the performance requirements, security, maintainability, portability etc.

3- Design:-

The goal of design phase is to transform the requirements specified in SRS documents into a structure that is suitable for implementation in some programming language. i.e the structured architecture is derived from the SRS documents.

Two design approaches are followed.

i) Traditional design Approach.

ii) Object-Oriented design Approach.

Traditional design Approach:-

It consist of two different activities.

a) Structured analysis

b) Structured design

a) Structure analysis:-

In this activity, the analysis of the requirement specification is carried out where the detailed structure of the problem is examined. During structured analysis the functional requirements specified in the SRS document are decomposed into subfunctions and the data flow among these subfunctions is analyzed and represented diagrammatically in the form of DFD.

Success needs no explanation. Failure has none.

Essential | Job to do | Phone No.

13 Thursday

b) Structure design:-

Once structured analysis activity is completed, the structure design activity starts. It undergoes two main activities.

i) Architectural design

ii) Detailed design.

Architectural design involves decomposing system into modules and representing the interfaces and the invocation relationship among the modules. It is also known as Software Architecture.

During detailed design, internals of individual modules are designed in greater detail i.e the data structures and algorithms of the modules are designed and documented.

ii) Object Oriented Design Approach:-

Here various objects that occurs in problem domain and solution domain are first identified and the different relationships that exist among these objects are identified. The object structure is further refined to get detailed design. It has several benifits like lower development time and effort and better maintainability.

4) Coding and Unit testing:-

Here the software design is translated into source code where each component of the design is implemented as a program module. Here the coding should be done by following some coding

Hundreds of businessmen have succeeded without a MBA degree, but none without common sense

Essential | Job to do | Phone No.

014-351
3rd Week

14 Friday

Standards of the Organization.

Each module that is then unit tested to determine correct working of all individual modules.

5) Integration And System Testing:

During this phase, the modules are integrated in a planned or incremental manner and during each integration step, the partially integrated system is tested. When all the modules are integrated, system testing is carried out to ensure that the developed system conforms to all requirements laid out in the SRS document.

System testing consist of three different types of testing activities.

i) α-testing:- It is the system testing performed by the development team.

ii) β-testing:- It is the system testing performed by a friendly set of customer.

iii) Acceptance testing:- It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered project.

6) Maintenance:

Maintance involves performing the following activities.

i) Corrective maintance:- Correcting errors that were not discovered during the product development phase.

Discussion is an exchange of knowledge. Argument is an exchange of ignorance.

Essential | Job to do | Phone No.

015-350
3rd Week

ii) Perfective Maintenance:- Improving the implementation of the system and enhancing the functionalities of the system according to the customer's requirements.

iii) Adaptive maintenance:- Porting the software to work in a new environment.

15 Saturday

Drawback:

→ If any type of error present in earlier phase then it couldn't rectify if because button to top movement is not possible.

→ It is suitable for long term Project.

→ It doesn't support different type of risk.

16 Sunday

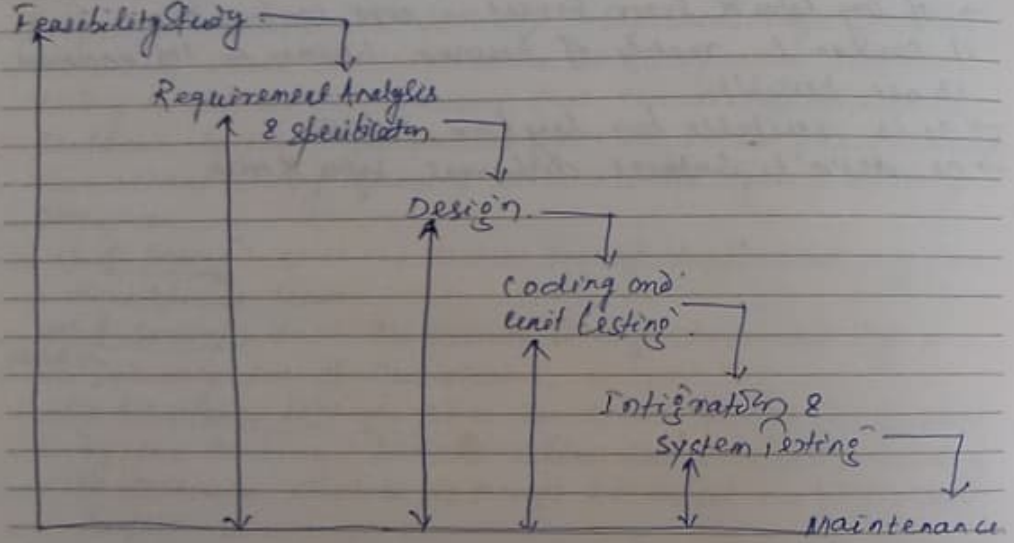
You cannot strengthen the weak by weakening the strong.

Essential | Job to do | Phone No.

17 Monday

Iterative Waterfall Model

This is a similar to classical waterfall model except there is a feedback from every phase to it's preceeding phase. It allows to correct the errors during a phase that we committed in later phase.



Phase Containment Error:

This is the principle of detecting errors as close to their point of introduction as possible. Phase containment minimizes the cost to fix errors.

Today is the future that you created yesterday

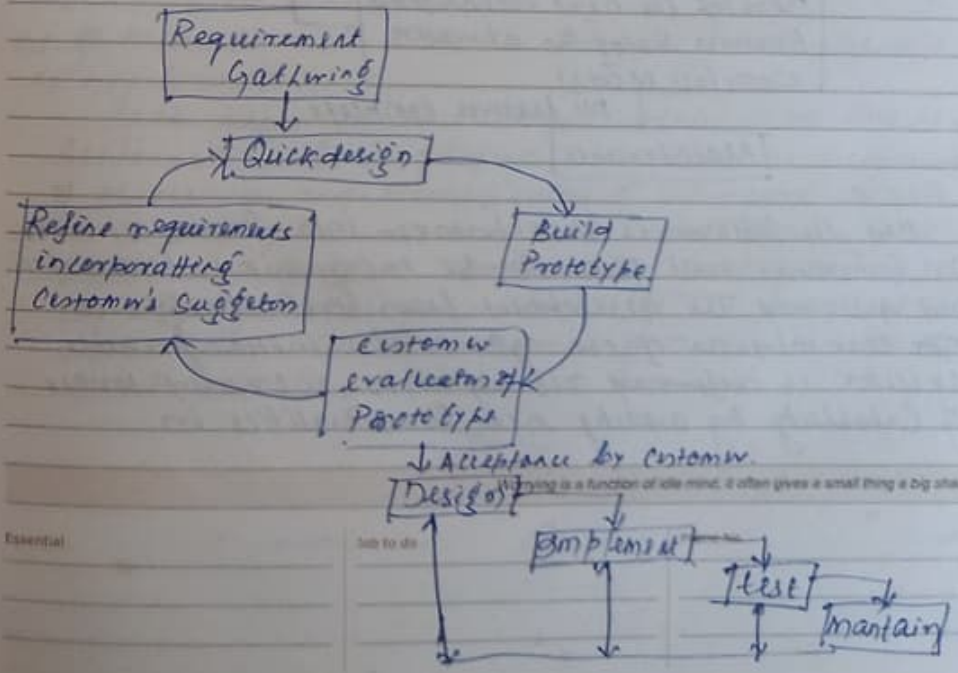
Essential | Job to do | Phone No.

18 Tuesday

PROTOTYPING MODEL

In this model product development starts with an initial requirements gathering phase. A quick design is carried out and the prototype is built. This prototype is submitted to customer for his evaluation. Based on the customers feedback and modifying the prototype continues till the customer approves the prototype. Then actual system is developed by using Waterflow Model.

So By constructing the prototype and submitting it to the customer, many customer requirements get properly defined and technical issues are resolved by experimenting with the prototype. This minimizes the change requests from the customer and the associated redesign cost.

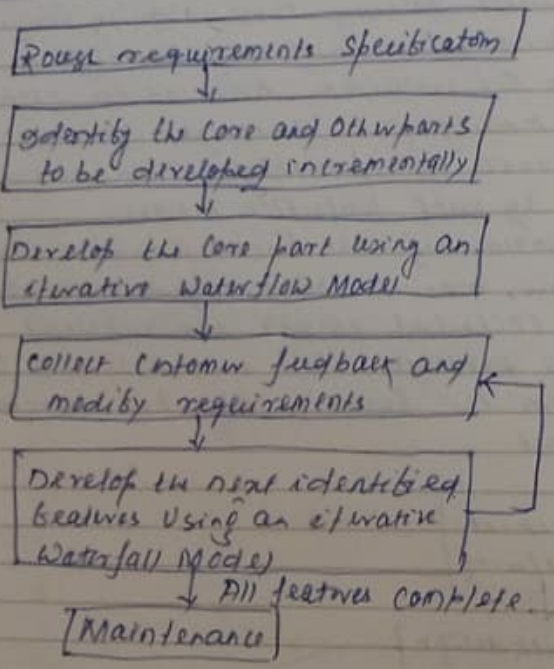


Essential | Job to do

Nothing is a function of idle mind, it often gives a small thing a big shadow

19 Wednesday

Incremental Model Or Evolutionary Model:



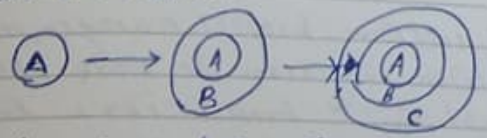
Here the software is first broaken into Several modules Or functional unit which Can be incrementally Constructed and delivered. The development team first develops the core modules of the System. This initial product skelter is refined into increased levels of Capability by adding new functionalities in

To get things done, choose a busy person. The other kind has no time.

Essential | Job to do | Phone No.

Successive versions.

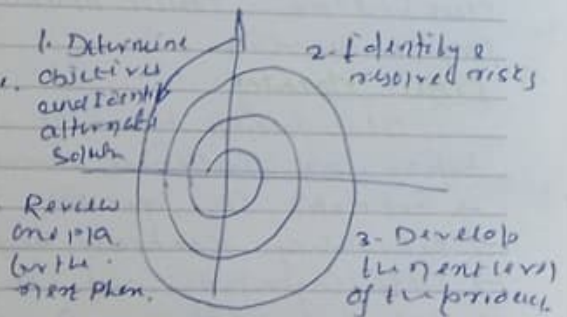
Thursday 20



A, B, C are modules of a software product that are incrementally developed and delivered.

Spiral Model

It is a Spiral in structure. The exact no. of loops is not fixed. Each loop of spiral represent a phase of the software process. Here, this Model is more flexible. Since exact no. of phases through which the product is developed is not fixed.



Each phase in this Model is split into four sectors. The first quadrant identifies the Determination of objectives and identification of alternative solutions. The second quadrant shows the identification and resolution of risk. The During 3rd quadrant Development the next level of the product is done and the at last Review and plan for the next phase is done.

It's always been that the horse does the work & the coachman is tipped.

Essential | Job to do | Phone No.

21 Friday

Software Project Management :-

The main goal of software project management is to enable a group of software engineers to work efficiently towards successful completion of project. For a large project the project manager takes all the responsibility.
The activities that the project Manager handles are broadly classified into Project planning, project Monitoring and control activities.

Project Planning :-

The project planning is undertaken and completed before any development activity starts. It includes
→ Estimate Cost, duration, Effort and Size.
→ scheduling manpower and other resources.
→ Staff Organization and staffing Plans.
→ Risk Identification, Analysis.
→ Quality assurance plan, Configuration management plan etc.
Out of all these measure project size estimation is the most fundamental parameters based on which all other planning activities are carried out.

Metrics for Project Size Estimation :-

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Two metrics are widely used for project size estimation
i) Lines of Codes (LOC)
ii) Function Point (FP)

022-343
8th Week

22 Saturday

i) **Lines of code :- (LOC)**

Using this metric, the project size is estimated by counting the number of source instruction in the developed program, ignoring the comment lines and headerlines.
To estimate LOC at the beginning of the project, the Project Manager divide the project into modules and each modules into sub modules and so on until the size of the different leaf-level modules can be approximately predicted. So by estimating leaf level modules, the project Manager arrives at the total size of the project. This activity needs the past experience of developing a project.

Drawback :-

1) LOC gives a numerical value of problem size that can vary widly with individual coding style.
2) LOC focuses on the coding activity alone, it doesn't consider the efforts needed in other phases of software life cycle.
3) LOC measure Co-relates poorly with the quality and efficiency of the code.
4) LOC metric penalizes use of high-level Programming language, code reuse etc.
5) LOC metric measures the lexical complexity of a Prog and doesn't address the issue of logical or structural complexity.
6) It is very difficult to accurately estimate LOC in the final product from the problem specification.

23 Sunday

24 Monday

ii) Function-Point Metric

The function-point metric can be used to easily estimate the size of the software product directly from the problem specification. Since the software product depends on the different function and features, the function point metric takes the following parameters into consideration.

a) Number of inputs
b) Number of outputs
c) Number of inquires
d) Number of files
e) Number of interfaces

Actually the Function point is computed in two steps.

1) Compute Unadjusted function point (UFP)
2) Compute Technical Complexity Factor (TCF)

The above parameters computes Unadjusted function point (UFP)

UFP = (No of input) *4 + (No. of output) *5 + (No. of inquires) *4 + (No. of files) *10 + (No. of interfaces) *10.

After UFP computed, TCF is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rate, throughput and response time etc. Each of these 14 factors are assigned a value from 0 to 5.

So the resulting numbers are summed to get total degree of Influence (DI)

So TCF = (0.65 + 0.01 * DI). where $0 \leq DI \leq 70$

So Finally FP = UFP * TCF.

Variety is the spice of life, but monotony supplies the food

Essential | Jobs to do | Phone No.

Drawback:-

i) It doesn't take the algorithmic complexity into account. so Another Metric Method known as Feature point Matric which incorporate an extra parameter into algorithmic complexity of a software including all other parameters of ~~Fore~~ supported by Function point Metric.

Project Estimation Technique:-

The parameters which are participated in project Estimation are project size, project cost, project duration and effort required to develop a project.

There are 3 basic project estimation Technique present.

i) Empirical estimation techniques.
ii) Heuristic technique.
iii) Analytical estimation technique.

26 Wednesday

Empirical estimation Technique:-

These techniques are based on making an educated guess of the project parameters which needs prior experience with the development of similar product. It include two technique. 1) Expert judgement Technique.
2) Delphi cost Estimation.

1) Expert judgement Technique:-

Here a group of experts makes an educated guess of the problem size after analysing the problem thoroughly. They estimate the cost of different components of the system

Another Method for finding cost estimation using Empirical Technique. By using historical data, the planner estimate an Pessimistic, Optimistic and Most Likely size value for each function. Then expected value is computed. so

$$EV = (S_{opt} + 4S_m + S_{pes}) / 6.$$

The best way to escape from a problem is to solve it

Essential | Jobs to do | Phone No.

A typical estimation model is derived using regression analysis on data collected from past software projects. So the total effort in person month is $E = A + B \times (EV)^C$ where A, B, C are const.
EV = estimated value (FP or LOC)

and then combines them to arrive at the overall estimates.

2) Delphi cost Estimation :-

It is carried out by a team composed of a group of experts and a co-ordinator. The co-ordinator provides each estimator with a copy of the software requirement specification document and a form for recording his cost estimate. Estimator completes their individual estimates and submit it to the co-ordinator. In their estimates, the estimator mention any unusual characteristic of the product which has influenced their estimation. The co-ordinator prepares and redistributes the summary of the response of all the estimator and includes any unusual rational noted by any of the estimators. Based on this summary, the estimate re-estimates. This process iterated for several round. After completion of several iteration of estimations, the Co-ordinator takes the responsibility of compiling the result and preparing the final estimation.

Note: There shouldn't be any discussion among estimators during estimation.

Heuristic Estimation Technique:-

It assumes that the relationship among the different project parameters can be modelled using some mathematical expression. Once the basic or independent parameter known then the other dependent parameter

can be easily determined.

Heuristic estimation Technique is of two type
1) Single variable Model (COCOMO Model)
2) Multi variable Model (intermediate cocomo Model)

Single variable Model :-

The single variable estimation Model is

Estimated Parameter = $C_1 \times e^{d_1}$

Where e = independent or basic parameter which is already calculated.

C_1 d_1 → Constants whose values depends on the data collected from past project.

Estimated Parameter :- dependent parameter whose value to be calculated like effort, project duration, staff size etc.

Constructive COST estimation MOdel :- (COCOMO)

COCOMO Model for cost estimation is formulated on three different categories based on the development complexity.

Organic
Semidetached
Embeded

Organic :- If a development project is organic if the project deals with developing a well understood application program and the development team members are experienced person.

Semidetached :- If a development project is semidetached if the development team consist of experienced and inexperienced staff and they have limited experience on related system.

29 Saturday

Embeded: A development project is Embeded type of the project is strongly coupled to complex hardware or stringent regulations on the operational procedure exist.

For calculating cost by COCOMO, two parameter we need.

i) KLOC (Kilo lines of source code)

ii) PM (One Person-Month) → It is the effort an individual can put in a month.

Basic COCOMO Model: (It computes cost as function of project size in estimated lines of code) It gives an approximate estimation of the project parameters

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{TDEV} = b_1 \times (\text{Effort})^{b_2} \text{ Months}$$

KLOC → estimated size of the software product in KiLo Lines of code

30 Sunday TDEV → estimated time to develop the software

Effort → total effort required to develop the Software Product

a_1, a_2, b_1, b_2 → Const. for each category of software product (Organic etc)

Given:-

Product type	a_1	a_2	b_1	b_2
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embeded	3.6	1.20	2.5	0.32

Knowledge comes from taking things apart. Wisdom comes from putting things together

Essential | Job to do | Phone No.

031-334 5th Week

31 Monday

29 COCOMO Model effort is also known as nominal effort and duration estimation is known as nominal duration estimation

Ex:- A Software Product is Organic type

~~Estimated Size of the~~ Estimated size of the product = 32,000 Lines of S-code

Avg. salary of Software Engineer = 15,000.00 PM

Find Effort required to develop the software product and the nominal development time.

Soln:-

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2}$$

For Organic, $a_1 = 2.4$, $a_2 = 1.05$

KLOC = 32

So Effort $= 2.4 \times (32)^{1.05} = 91$ PM

Nominal development Time $= b_1 \times (\text{Effort})^{b_2}$

here $b_1 = 2.5$, $b_2 = 0.38$

so Nominal development Time $= 2.3 (91)^{0.38} = 14$ months

So the cost required to develop the product is

$14 \times 15{,}000 = \text{Rs } 210{,}000.00$

Analytical technique for ~~two~~ Software Estimation:-

It derive the project parameter starting with basic assumptions regarding the project. It is based on some scientific tools.

HALSTEAD'S SOFTWARE SCIENCE:- It is a one type of analytical technique to measure size, development effort, and development cost of software products.

Remember that overnight success usually takes about fifteen years

Essential | Job to do | Phone No.

01 Tuesday

Halstead uses some primitives programming parameters to developed expression for different parameters.

For a given program let.

η_1 → no. of unique operator

η_2 → no. of unique operand.

N_1 → Total no. of operator

N_2 → Total no. of operand.

Guideline for identifying different operator & operand.

Operator:-

→ assignment, arithmetic and logical operator

→ a pair of parenthesis ({} or () or []) are treated as single operator

→ label is as an operator if it is a target of GOTO

→ if--then--else--endif, while--do are single operator each.

→ ; , are treated as operator

→ Function name in function call statement is a operator

→ In function declaration the function name are not taken as operator

→ tb

Operand

→ variables and constants which are used in operators in expression are operand.

→ The argument list in a function or a function call are operand.

→ the parameter list in function declaration are not operand.

→ the argument in function declaration are not operand.

Ex:- func(a,b);

here func, (,) are operator and a & b are operand.

but

```
int func (int a, int b)
  {
  :
  }
```

here (), {} are operator, no operand.

02 Wednesday

i) Program length :-

Total no. of operator and operands used in a program. i.e program length $= N = N_1 + N_2$

ii) program vocabulary :- The total no. of distinct operator & operand. So

program vocabulary $\eta = \eta_1 + \eta_2$.

iii) program volume :- The volume V is the minimum no. of bits needed to encode the program.

so prog. volume $= V = N \log_2 \eta$.

iv) Effort estimation :-

Effort to develop a prog. is obtained by deriding the prog. volume by level of prog. lang. used to develop the code.

So $E = V/L$.

so programming effort $= E = V^2/V^*$ where $L = V^*/V$

where $V^* = (2+\eta_2) \log (2+\eta_2)$

Or iv) Programming effort $E = D * V$.

Where D = Difficulty

$= \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$

03 Thursday

E = the no. of mental discriminations required to implement the prog.
or (Effort) required to read & understand the prog.
e = Programming effort.
L = Program level / Abstraction measure the level of abstraction in prog. Language
V = prog. volume
V* = potential minimum volume. Which is a volume of the most succinct prog. in which prog can be coded.

The programmer time
$T = E/S$
where E = Programming effort.
S = Speed of mental discrimination (usually 18)

So Estimated length of a program
$= N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

Ex:- find the Estimated length and volume of the following prog.

```
main()
{
 int a,b,c, avg;
 scanf("%d %d %d", &a,&b,&c);
 avg = (a+b+c)/3;
 printf("avg=%d", avg);
}
```

Coming together is beginning. Keeping together is progress. Working together is success.

04 Friday

Sol":-
The unique operators are
main, (), {}, int, scanf, &, ,, ;, =, +, /, printf
So $\eta_1 = 12$.
unique operands are
a, b, c, &a, &b, &c, a+b+c, avg, 3, "%d %d %d", "avg = %d"
So $\eta_2 = 11$
So vocabulary $= \eta_1 + \eta_2 = 12+11 = 23$
~~So~~ So program length $= N = N_1 + N_2$
$N_1 = 27$
$N_2 = 17$
So $N = 44$ → not required
Now Estimated length
$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
$= 12 \log_{12} + 11 \log_{11} = 81$
Volume $= N \log \eta = 81 \log 23 = 366$

NOTE:- ——— x ———
The intermediate COCOMO model:- It computes the cost as a function of prog. size and a set of "cost drivers", such as hardware, personnel, project attributes, product attributes
So the effort in human months can be estimated as
$E = a_i (KLOC)^{b_i} * EAF$

Be prepared. You never get a second chance to make a good first impression.

05 Saturday

Where E = the effort applied in person Month.
EAF = effort Adjustment factor. ranges from 0.9 to 1.4
a_i & b_i = Const. for different project.

Software project	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20.

The Advanced COCOMO Model:-
It incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step of the software process.

So

$$E = \left[\frac{LOC \times B^{0.333}}{P}\right]^3 \times \frac{1}{t^4}$$

06 Sunday

Where E = effort in person-months or person years
t = Project duration in months or year,
B = special skill factor
for small proj (KLOC = 5-15), B = 0.16
for large proj (KLOC > 70), B = 0.39
P = productivity parameters
= 2000 for real time embedded software
= 10,000 for telecommunication system.
= 28,000 for business application system.

Planning process or task:
The following are the steps in the to project planning process

i) Identify the tasks
ii) Clearly state the objectives of each task
iii) Estimate the personnel, Time and Resources to meet objectives
iv) Develop a task sequence
v) Estimate the product development costs

i) Identify the task:-
→ The task bring the project from a problem to a product ~~where critical activities~~
→ initially activities to be performed
→ start with major development activities
→ Break down into smaller task

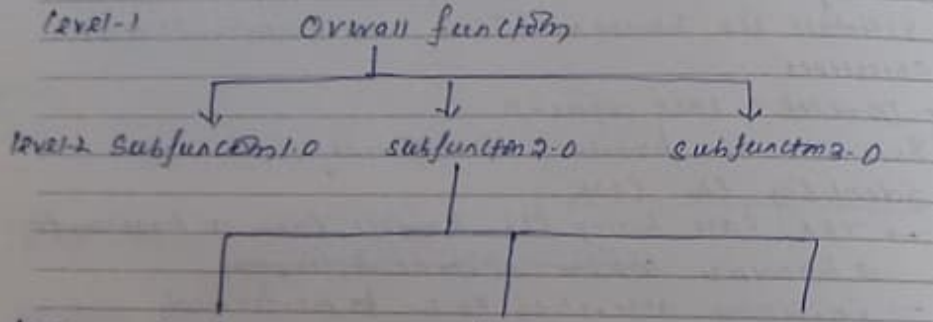
ii) Clearly state the objective of each task:-
The tasks refined to show results of the activities are stated objectives.
→ Easily understood by entire design team
→ Specify as to what information is to developed
→ It should be feasible

iii) Estimate the personnel, time and resources to meet the objectives:
A variety of schemes are used to estimate, all are based on assumptions. Track records within Companies is always a best indicator.

07

08 Tuesday

Reasonable estimates of time are based on Product complexity
Function diagram for Product complexity.



So product complexity $PC = \sum j \times F_j$

Where j = level in function diagram.
F_j = number of functions at that level

Time Estimation:
$$Time(hours) = A \times PC \times D^{0.85}.$$
Where PC = Product complexity.
A = Constant based on company history & size
= 30, for small and inexperience.
= 150, for large with normal experience.

Spend your life lifting people up, not putting people down

Essential | Job to do | Phone No

09 Wednesday

D = Project difficulty
= 1 if Not difficult, know technology.
= 2 if Difficult, some new technology.
= 3 if very difficult, lots of new technology.

A Simple Weighted Avg. technique:
$$\text{Time Estimate (PERT)} = \frac{Opt + (4 \times Likly) + Pess}{6}$$

iv) Develop a task sequence:-
Develop a sequencial tasks sequence and parallel tasks sequence and then develop a planning or scheduling chart which may be milestone or Gantt Chart or PERT.
v) Estimate the product development cost.

Planning and Scheduling:-

For the scheduling different charts are available.
i) Gantt Chart.
ii) Milestone Chart.
iii) PART Chart

Judge people from where they stand, not from where you stand

Essential | Job to do | Phone No

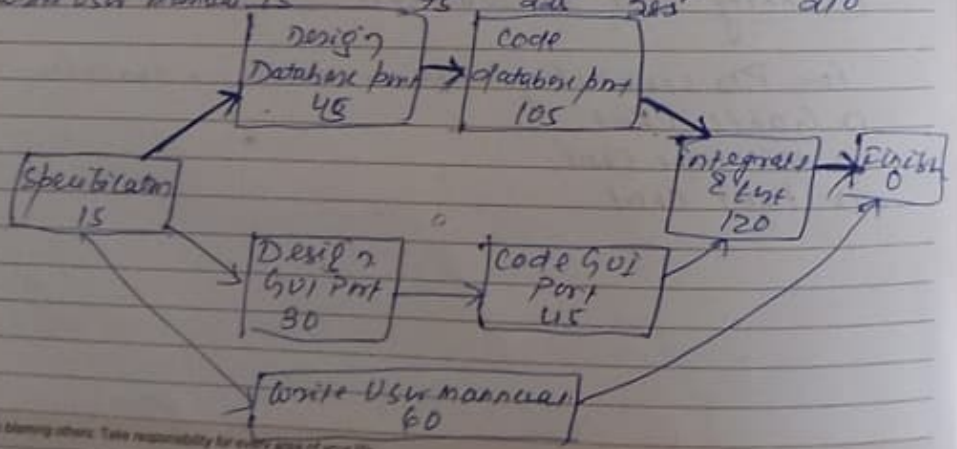
10 Thursday

GANTT Chart:-

It is mainly used to allocate resources to activities. It is special type of bar chart where each bar represents an activity. The bars are drawn along the timeline. The length of each bar is proportional to the duration of the time planned for the corresponding activity.

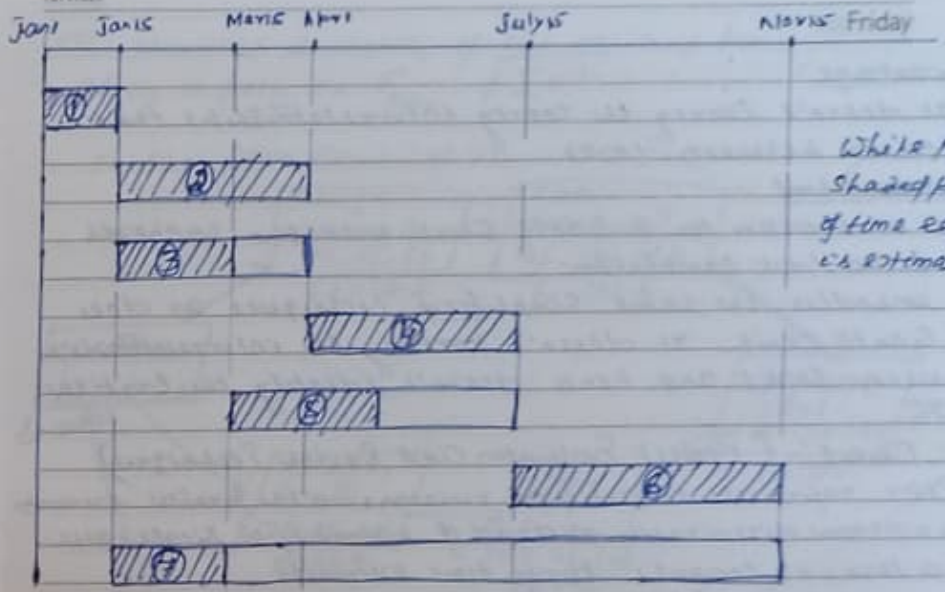
Consider the following table

Task	ES	EF	LS	LF	ST
1- Specification Part	0	15	0	15	0
2- Design database Part	15	60	15	60	0
3- Design GUI Part	15	45	90	120	75
4- Code Database Part	60	165	60	165	0
5- Code GUI Part	45	90	120	165	75
6- Integrate & Test	165	285	165	285	0
7- Write User Manual	15	75	225	285	210



(MIS Prob)

11 Friday



(Gantt Chart of MIS Problem)

Advantage:-

i) Direct co-relation with time.
ii) Straight forward relationship with projects involving a limited no of tasks.
iii) Straight forward integration of subtasks having separate scheduling chart.
iv) Time scheduled is flexible and is expanded to show tasks of shorter nature.
v) Progress against the plan is easily reflected.

12 Saturday

Disadvantage:
i) It doesn't convey the convey interrelationships that may occur between tasks.

Milestone Chart:
→ It is similar to a Gantt Chart with the emphasis placed on task completion
→ It embodies the same simplistic techniques as does the Gantt Chart. It doesn't portray the interrelationship between tasks and hence doesn't identify the critical path.

PERT Chart: (Project Evaluation And Review Technique)
PERT represents the statistical variations in the project estimates, through a three time estimation instead of estimating a single value for each task, it computes three time estimates
i) Pessimistic time (t_p) → The time the activity would take if things did not go well (longest time)
ii) Most likely time (t_m) → the best estimates of activity duration
iii) Optimistic time (t_o) → the time the activity would take if things did go well (shortest time)

So the weighted avg is used to establish
Expected Time = $t_e = \frac{t_o + 4t_m + t_p}{6}$

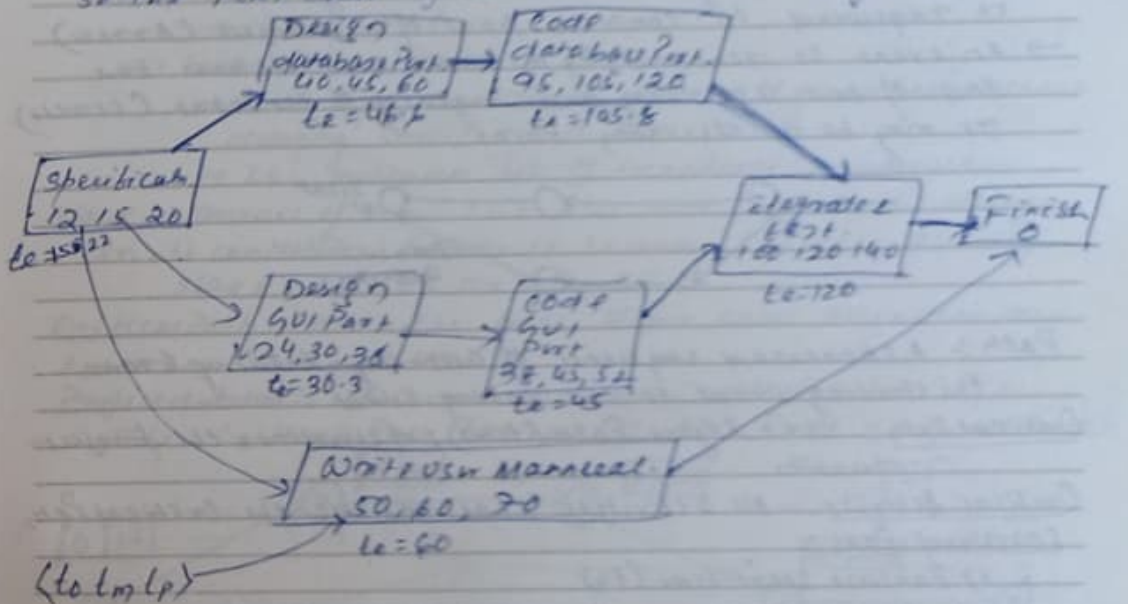
The standard deviation of the expected time for each activity $\delta = \frac{t_p - t_o}{6}$

Course | Job to do | Phone No.

Monday 14

The standard deviation of the expected time along a path is $\delta_{path} = \sqrt{\sum \delta^2}$

So the PERT Chart of the MIS Problem is given as

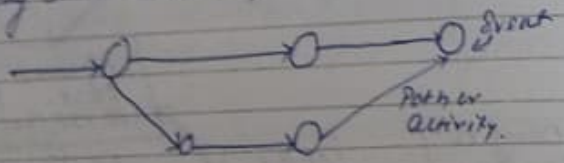


Critical Path: [illegible]

Essential | Job to do | Phone No.

15 Tuesday

CPM (Critical Path Method)
→ It has some common characteristics with PERT.
How an activity is a time-consuming effort that is required to complete part of a project (Arrow)
→ An event is denoted by a circle and defines the end of one activity and beginning of the next. (Circle)
It may be a decision point.

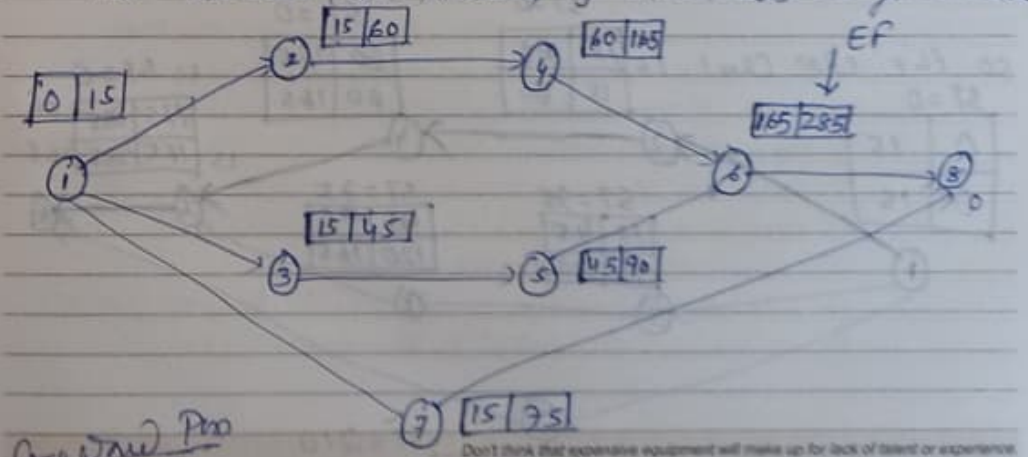


Path :- A connected sequence of activity leading from the starting event to the ending event.
Critical Path :- The logest Path (time); determines the project duration.
Critical Activity :- All activities that make up the Critical Path.
Forward Pass :-
i) Earliest start time (ES)
→ It is the earliest time an activity can start.
→ ES = Maximum EF of immidiate Predicessor.
ii) Earliest Finish time (EF)
→ earliest time an activity can finish
→ EF = ES + t, t = time duration.

Name | Job to do | Phone No.

16 Wednesday

Backward Pass :-
i) Latest start Time (LS)
→ Latest time an activity can start without delaying Critical path time.
→ LS = LF - t.
ii) Latest finish Time (LF)
→ latest time an activity can be completed without delaying Critical path time.
→ LS = minimum LS of immidiate predecessor.
Slack :- amount of time this this activity can be delay in its completion before it becomes Critical activity.
Slake = LS - ES or LF - EF
Critical Path :- It is the path where each activities and event has zero slack.
Project duration :- Minimum project completion time.
The Critical Path Method of this Prob is given below

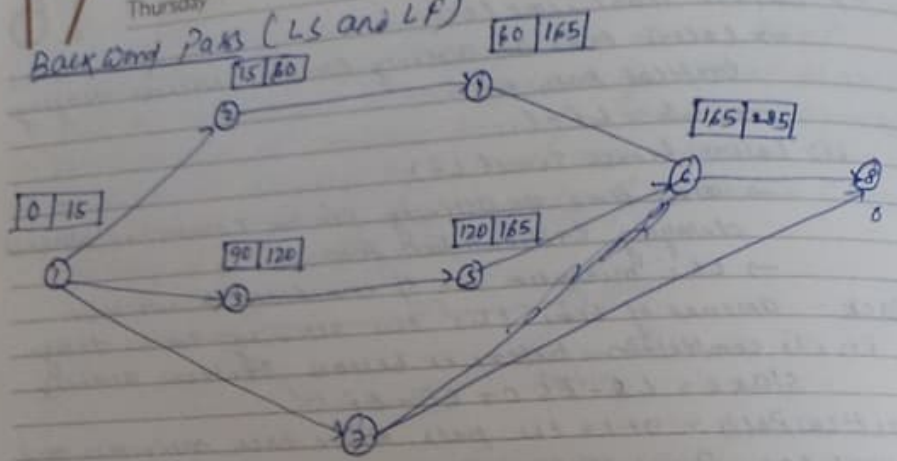


Forward Pass (ES, EF)

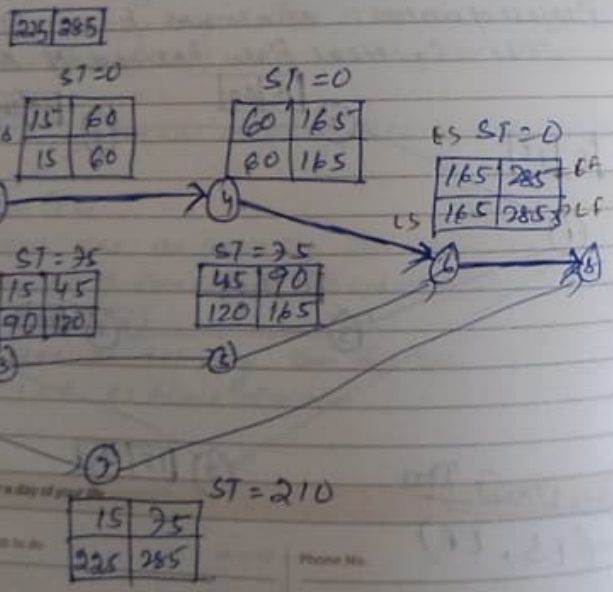
Name | Job to do | Phone No.

17 Thursday

Backward Pass (LS and LF)



(ST = Slack Time)



So the CPM Chart is

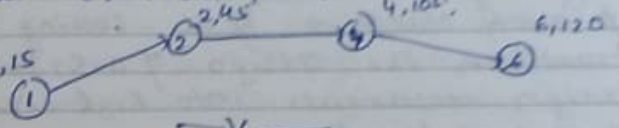
Today is important because you are exchanging it for a day of your life

Priorities | Job to do | Phone No.

049 316
Rin Week

18 Friday

Since the slack time is zero for every 1, 2, 4, 6 so this is the critical path.



Y

The full use of today is the best preparation for tomorrow

Essential | Job to do | Phone No.

FEBRUARY Coding and testing 2011

19

Once the design document has been prepared, then coding phase is going to be carried out. The coding phase is many challenging of the coding phase is to translate the design of a system written in design documents into high level language code and to unit test the code.

Each Organization has it's coding standards which is to be followed by the developer during coding. This is because.

→ A coding standard gives a uniform appearance to the codes written by different Engineers
→ It provides sound understanding of the code.
→ It encourages good programming practice.

Coding standard :-

1) Limiting the use of Global :- What type of data can be declared as global
2) The contents of the header preceding the code should have the following data.
   a) name of the module
   b) Date of the module Created.
   c) Author's name
   d) Modification history
   e) Synopsis of the modules
   f) Global variables
   g) Different functions supported.

Monday 21

3) Naming convention :-
   Global variables start with Capital letter.
   local variables are small case letter.
   constant name are always capital letter.
4) Error Return convention any exception handling mechanism :-

Coding Guidelines :-

1) Do not use a coding style that is too clever or too difficult to understand.
2) Avoid Obscure side effects.
3) Do not use identifiers for multiple purposes
4) Each variable should have descriptive name describe the purpose.
5) Code should be well-documented
6) The length of the any function should not exceed 10 source lines
7) Do not use goto statements

After coding has been done, the code review is carried out. The prime objective is to detect as many errors as possible.

Code Review :-

It is of two types.

FEBRUARY 2011

053-312
7th Week

22 Tuesday

i) Code Walk-Throughs
ii) Code Inspection

Code Walk-Through:-

This is a informal Analysis technique which is used to discover the algorithmic and logical error in the code.

Here the code is given to the each member of Walk-through team. The members do some test cases and simulate executions of the code by hand. And list the errors and exceptions found in the code.

Code-inspection:-

The aim of the code inspection is to discover errors caused due to oversight and improper programming.

The following are some errors to be checked during code inspection

a) Jumps into loops
b) Use of uninitialized variables
c) Non-terminating loops
d) Array index out-of-bounds.
e) improper storage and deallocation.

In business the man who doesn't read a book is usually selected to be one

Essential | Job to do | Phone No.

S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 • • •

054-311
8th Week

Software Maintenance

Wednesday 23

Software Maintenance is 3 types.

1) Corrective:- When the system needs to rectify the bugs, it is Corrective Maintenance.

2) Adaptive:- When the platform changes or some hardware changes, then Adaptive Maintenance is needed to run the software in the new platform.

3) perfective:- When the customer needs, the software should support new functionalities. Or some other request of the customer, then perfective Maintenance is used.

Legacy System and it's Maintenance:-

A legacy software are those software which are hard to maintain ~~are~~ to because of poor documentation, unstructured code, non-availability of personnel who are knowledgable on the product.

The legacy systems are were developed long time back. But now the recently developed system having poor design and documentation can be categorized as legacy system.

Software Reverse Engineering:-

It is a process of recovering the design and the requirement specification of a product from analysis of it's code.

Tommorrow is often the busiest day of the week it has to take care of everything

Essential | Job to do | Phone No.