LEARNING OUTCOMES

- Understanding the problem and corresponding requirement for development of software.
- Phases of the system development life cycle.
- develop data flow diagrams
- Perform a feasibility study of the system.
- Write detailed design specifications for programs and database.
- Select methods for evaluating the effectiveness and efficiency of a system.
- Apply different testing techniques on simple program.

CHAPTER 1

1. INTRODUCTION

The term software engineering is composed of two words, software and engineering.
Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.
Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.
So, we can define software engineering as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is efficient and reliable software product.Software engineering is application of a systematic and disciplined approach to the development, operation and maintenance of software in an efficient and cost effective way.

1.1 Programmes v/s Software product

Programmes: A computer program is a sequence of instructions, written to perform a specified task with a computer. A collection of computer programs and related data is referred as software.

Software program: there are three basic entities which are generally used while defining the principles of software engineering are –

1. Software process
2. Software projects
3. Software products

1.2 Difference between programmes and software product

- Programmes are set of instructions for computer to perform a specified task.
- Programmes are developed by individuals.
- Programmes are smaller in size.

• In case of programmes, programmer himself is the sole user. • In programmes, very little documentation is required. • Software product is merchandise consisting of a computer program that is offered to sale.	• Software products are developed by multiple users. • Software products are extremely large in size. • In software products, most users are not involved with development. • Software products must be well documented.

1.3 Emergence of software engineering

1. Early computer programming:- early commercial computer were very slow and too elementary as compared to today's standards. In 1968, key concept of modularity and information hiding were also introduced to help programmer's deals with every increasing complexity of software system.

2. High level programming:- the high level language that are more powerful, easier to use and directed towards specialized classes of problems. A list of some high level language:- COBOL, PL/I, BASIC, PASCEL, C, C++, JAVA, FORTAN

1.4 SOFTWARE DESIGN

Software design process has two levels:-

1. System design or external design
2. Internal design or detail design

There are basically three approaches to software design-

- Data structured oriented design
- Control flow based design
- Object oriented design

CHAPTER 2

2. LIFE CYCLE MODEL

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

2.1 THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure. A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

Different software life cycle models

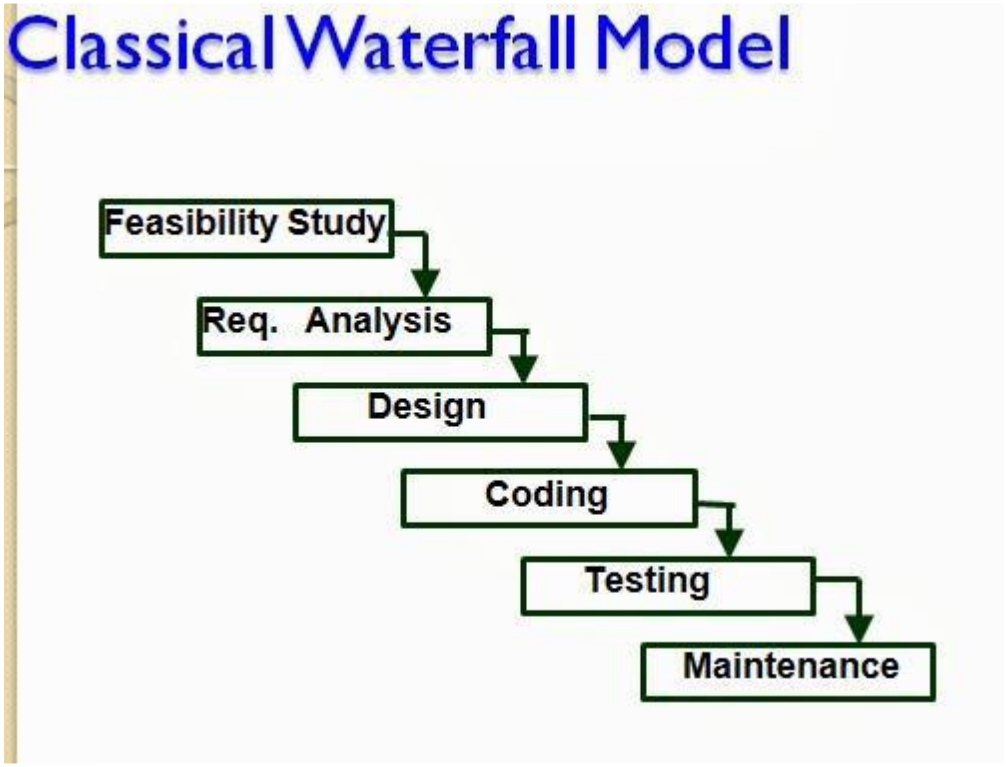
Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

- Classical Waterfall Model

- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

2.1.1 CLASSICAL WATERFALL MODEL

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a theoretical way of developing software. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases



Feasibility study - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.

- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

Requirements analysis and specification: - The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

Design: - The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

- Traditional design approach -Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.
- Object-oriented design approach -In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and unit testing:-The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each

component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.
Integration and system testing: -Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

- α – testing: It is the system testing performed by the development team.
- β –testing: It is the system testing performed by a friendly set of customers.
- 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 product.

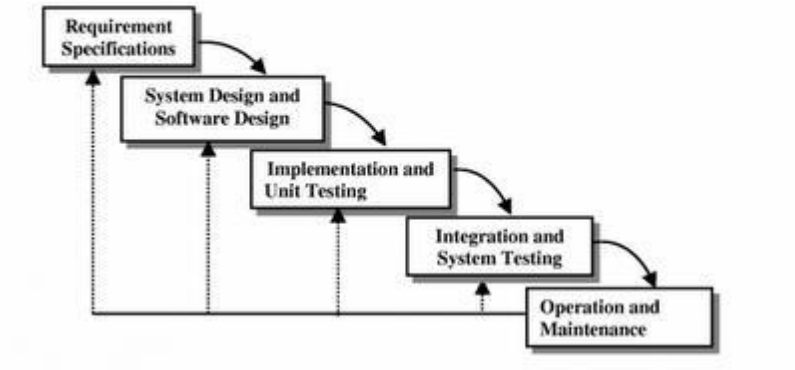
System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.
Maintenance: -Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratios. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

Shortcomings of the classical waterfall model
The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

2.1.2 ITERATIVE WATERFALL MODEL

To overcome the major shortcomings of the classical waterfall model, we come up with the



iterative waterfall model.

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

2.1.3 PRTOTYPING MODEL

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

Need for a prototype in software development

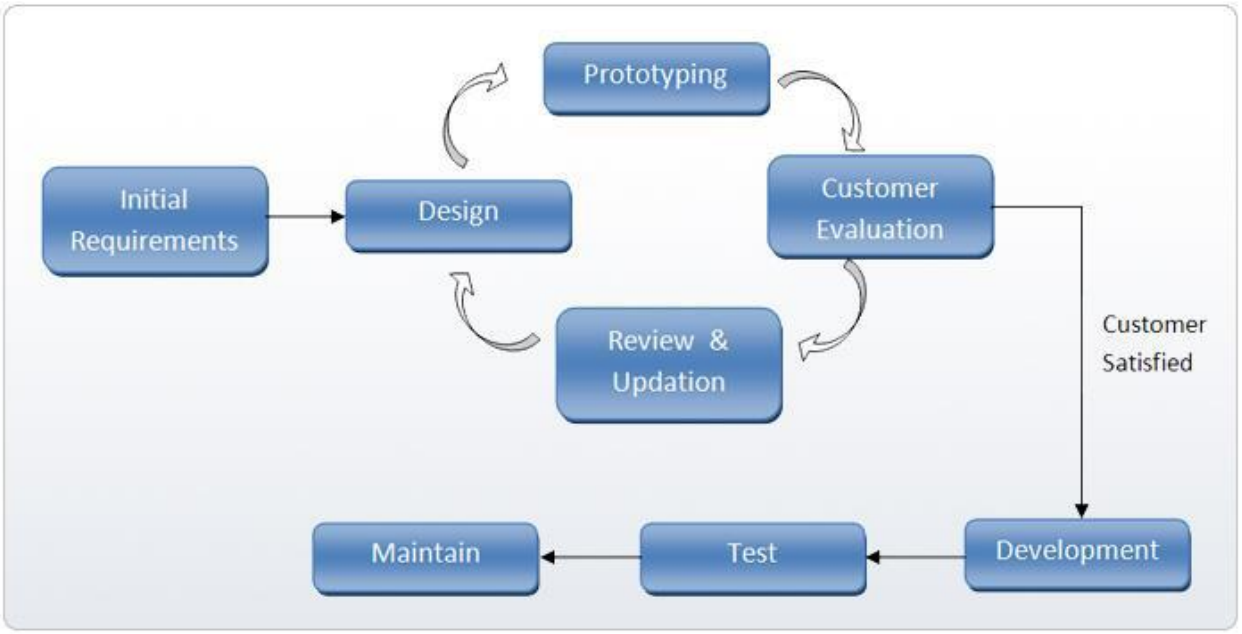
There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response

time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.
A prototype of the actual product is preferred in situations such as:
• User requirements are not complete
• Technical issues are not clear



2.1.4 EVOLUTIONARY MODEL

It is also called successive versions model or incremental model. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.
Applications:

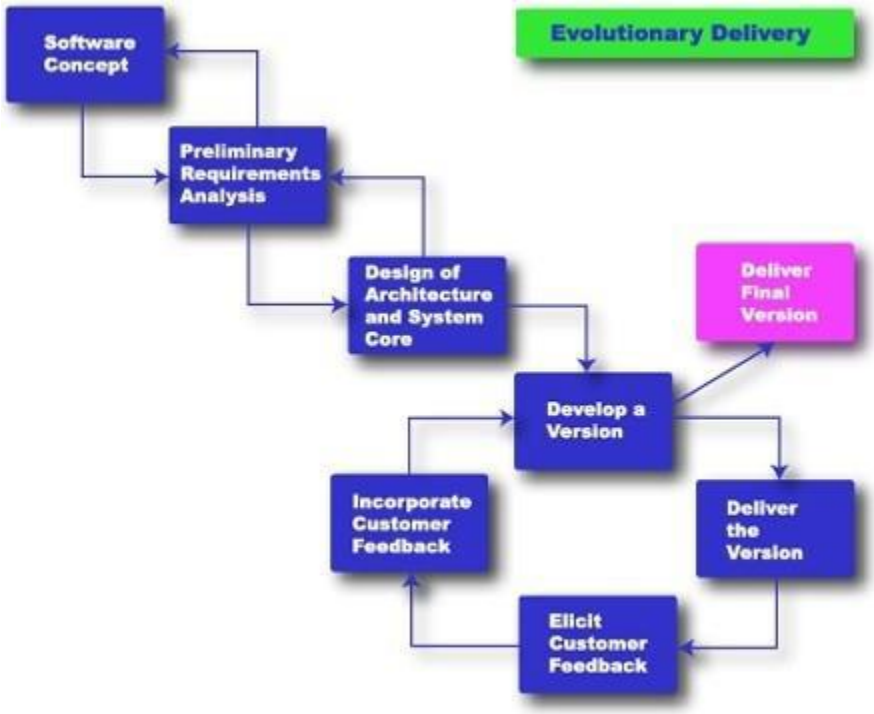
- Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.
- Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Advantages:

- User gets a chance to experiment partially developed system
- Reduce the error because the core modules get tested thoroughly.

Disadvantages:

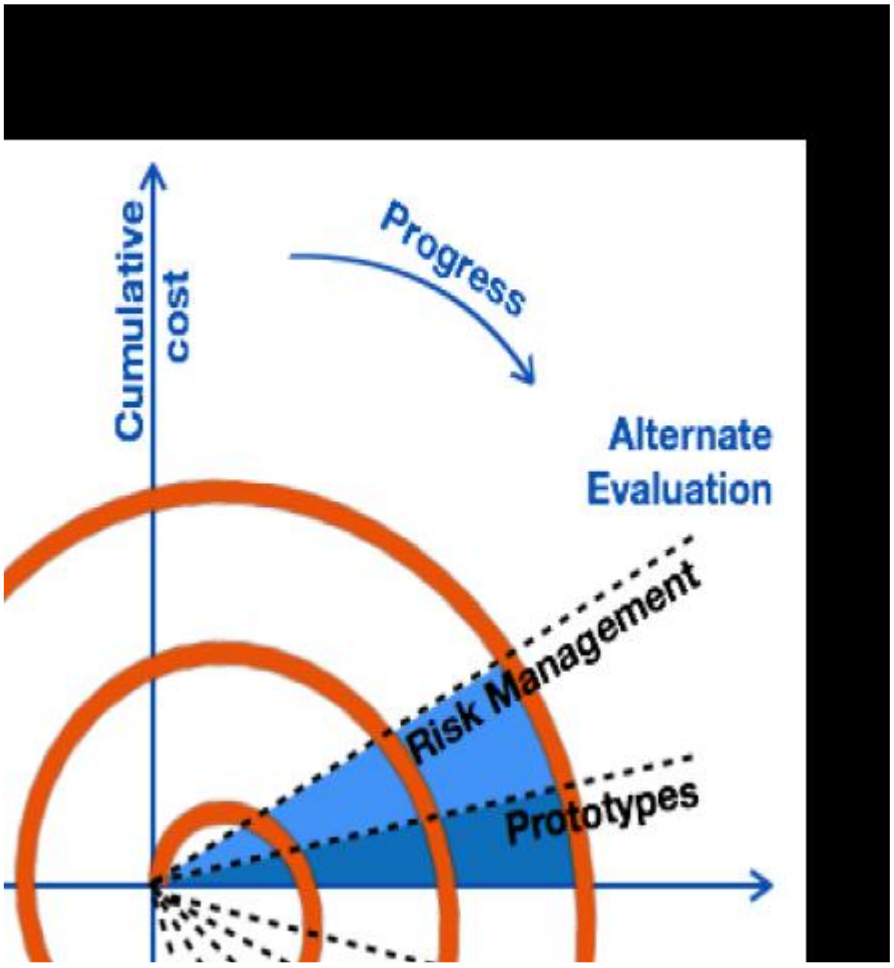
- It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.



2.1.5 SPIRAL MODEL

The Spiral model of software development is shown in fig. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four

sectors (or quadrants) as shown in fig. 4.1. The following activities are carried out during each phase of a spiral model.



First quadrant (Objective Setting)
• During the first quadrant, it is needed to identify the objectives of the phase.
• Examine the risks associated with these objectives.
Second Quadrant (Risk Assessment and Reduction)
• A detailed analysis is carried out for each identified project risk.
• Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
Third Quadrant (Development and Validation)
• Develop and validate the next level of the product after resolving the identified risks.
Fourth Quadrant (Review and Planning)
• Review the results achieved so far with the customer and plan the next iteration around the spiral.
• Progressively more complete version of the software gets built with each iteration around the spiral.
Circumstances to use spiral model
The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

2.2 Comparison of different life-cycle models

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

CHAPTER 3

3. Responsibilities of a software project manager

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

3.1 Skills necessary for software project management

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

3.2 Project planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

• Estimating the following attributes of the project:

Project size: What will be problem complexity in terms of the effort and time required to develop the product?

Cost: How much is it going to cost to develop the project?

Duration: How long is it going to take to complete development?

Effort: How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

• Scheduling manpower and other resources

• Staff organization and staffing plans

• Risk identification, analysis, and abatement planning

• Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

3.3 Metrics for software project size estimation

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product. Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

3.3.1 Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored. Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

3.3.2 Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.

The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the issue book feature (as shown in fig. 11.2) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP). UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance. Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input.

For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee. Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input. Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system. Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files. Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc. Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, FP=UFP*TCF.

3.4 COCOMO MODEL

Organic, Semidetached and Embedded software projects

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm's [1981] definition of organic, semidetached, and embedded systems are elaborated below.

Organic: A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a1 x (KLOC)a2 PM

Tdev = b1 x (Effort)b2 Months

Where

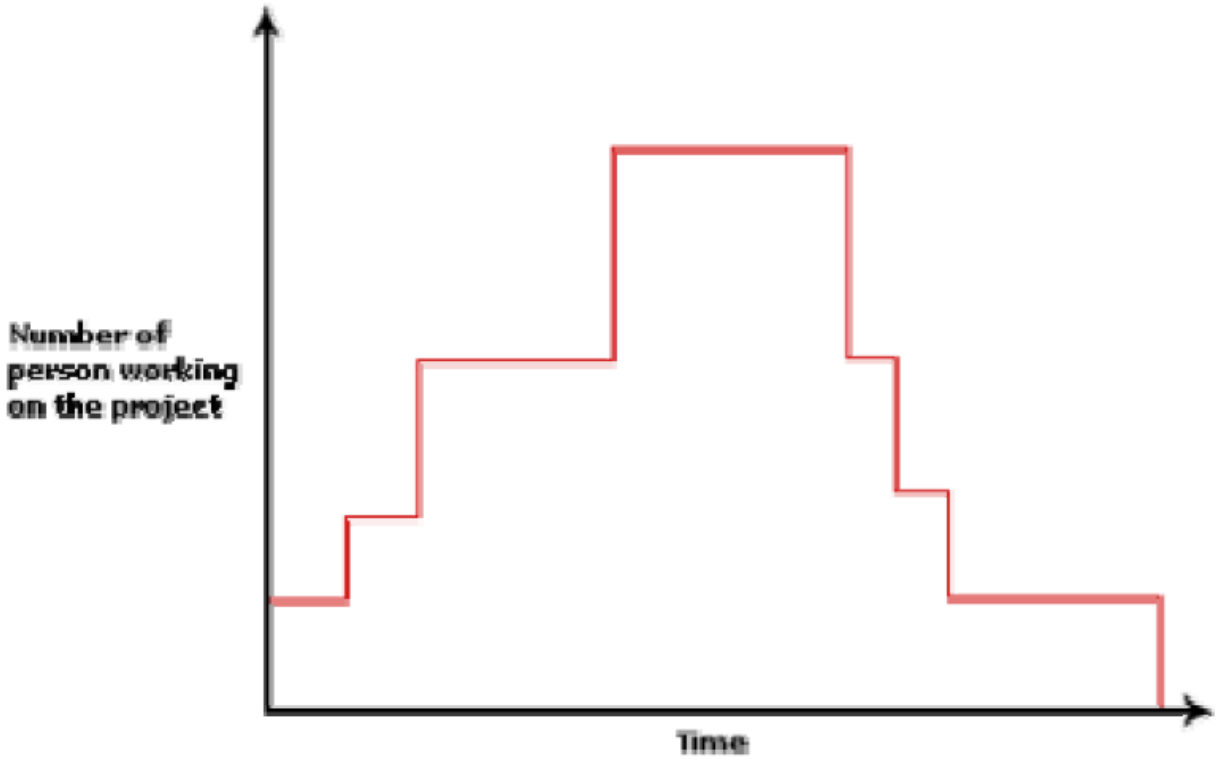
• KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

• a1, a2, b1, b2 are constants for each category of software products,

• Tdev is the estimated time to develop the software, expressed in months,

• Effort is the total effort required to develop the software product, expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in fig. 33.1). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve



According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a1, a2, b1, b2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic: Effort = 2.4(KLOC)1.05 PM

Semi-detached: Effort = 3.0(KLOC)1.12 PM

Embedded: Effort = 3.6(KLOC)1.20 PM

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

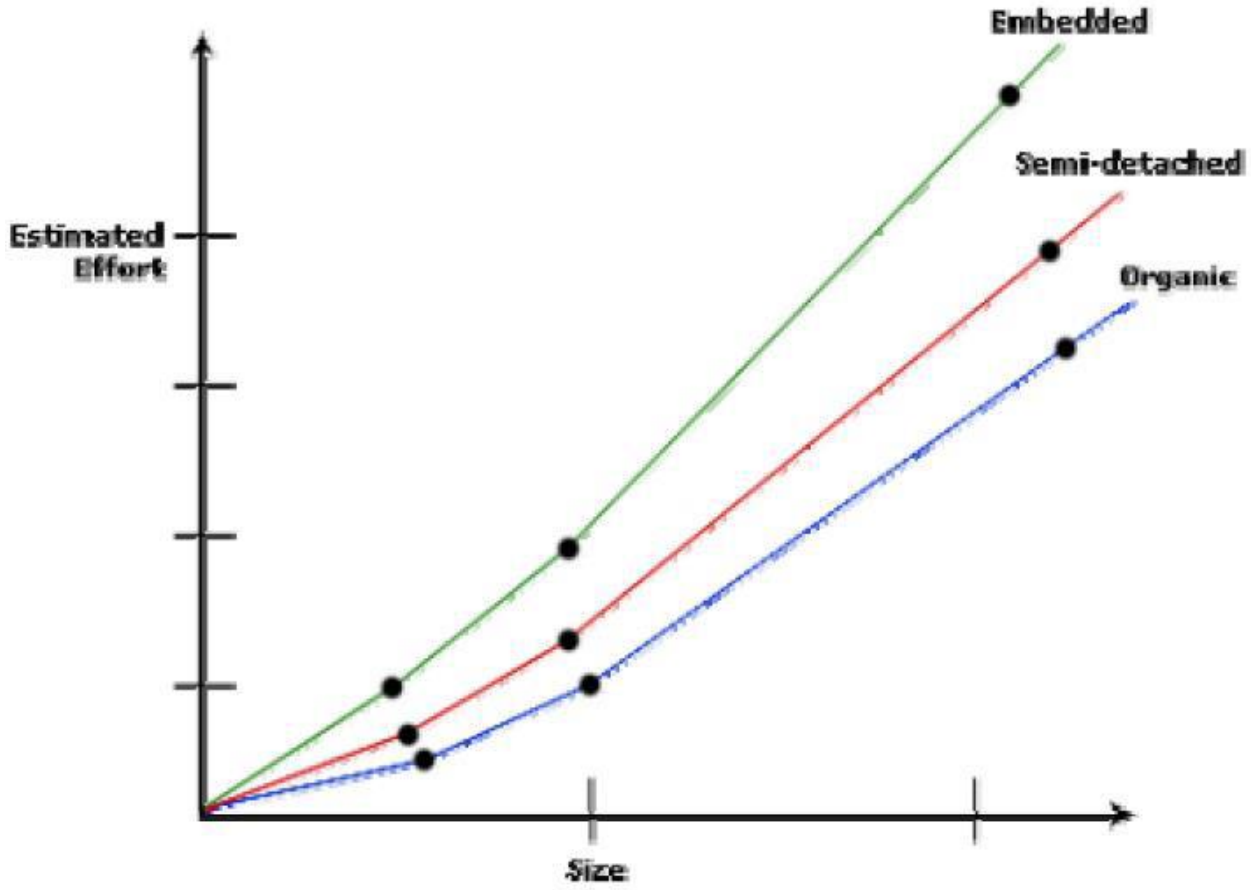
Organic: Tdev = 2.5(Effort)0.38 Months

Semi-detached: Tdev = 2.5(Effort)0.35 Months

Embedded: Tdev = 2.5(Effort)0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. 33.2 shows a plot of estimated effort versus product size. From fig. 33.2, we can observe that the effort is somewhat super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly

with project size.



The development time versus the product size in KLOC is plotted in fig. 33.3. From fig. 33.3, it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig. 33.3, it can be observed that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

CHAPTER 4

4. REQUIREMENTS ANALYSIS AND SPECIFICATION

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as system analysts.

The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

• 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 are the data output by the system?
• What are the likely complexities that might arise while solving the problem?
• If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness. When the analyst detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.

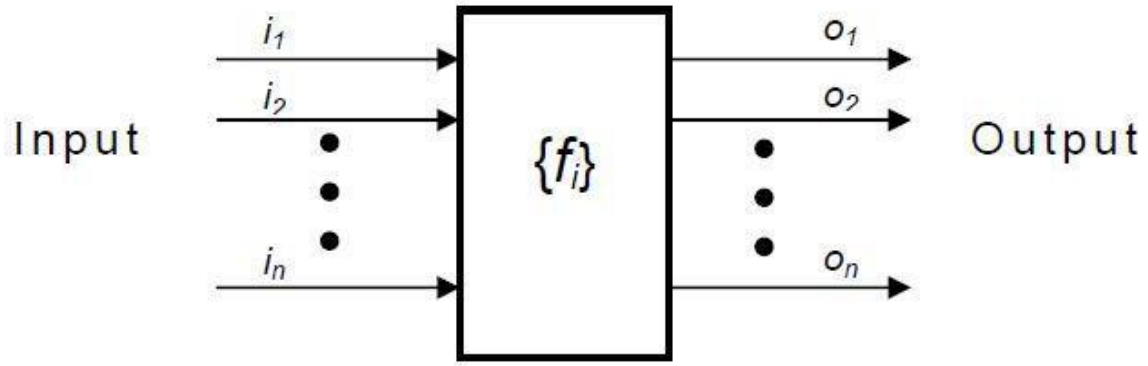
Parts of a SRS document

• The important parts of SRS document are:

Functional requirements of the system
Non-functional requirements of the system, and
Goals of implementation

4.1 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. 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.



4.2 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.

4.3 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.

4.4 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.

Example: - Consider the case of the library system, where –

F1: Search Book function

Input: an author's name

Output: details of the author's books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Documenting functional requirements

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash

is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.
Example: - Withdraw Cash from ATM
R1: withdraw cash
Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash; otherwise it generates an error message.

R1.1 select withdraw amount option
Input: "withdraw amount" option
Output: user prompted to enter the account type
R1.2: select account type
Input: user option
Output: prompt to enter amount
R1.3: get required amount
Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.
Output: The requested cash and printed transaction statement.
Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed

4.5 PROPERTIES OF A GOOD SRS DOCUMENT

The important properties of a good SRS document are the following:

- Concise. The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

- Structured. It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

- Black-box view. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

- Conceptual integrity. It should show conceptual integrity so that the reader can easily understand it.

- Response to undesired events. It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

- Verifiable. All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

4.6 PROBLEMS WITHOUT A SRS DOCUMENT

The important problems that an organization would face if it does not develop a SRS document are as follows:

- Without developing the SRS document, the system would not be implemented according to customer needs.
- Software developers would not know whether what they are developing is what exactly required by the customer.
- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
- It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

4.7 PROBLEMS WITH AN UNSTRUCTURED SPECIFICATION

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent.

CHAPTER 5

5.SOFTWARE DESIGN

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

- Architectural Design - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- High-level Design- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system

along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

- Detailed Design- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

5.1 MODULARIZATION

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction ca n be brought in the program
- Components with high cohesion can be re-used again.
- Concurrent execution can be made possible
- Desired from security aspect

5.2 CONCURRENCY

Back in time, all software's were meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, software has multiple modules, and then only one of all the modules can be found active at any time of execution.
In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.
It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.
Example
The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

5.3 COUPLING AND COHESION

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.
Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

- Co-incidental cohesion - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- Logical cohesion - When logically categorized elements are put together into a module, it is called logical cohesion.
- Temporal Cohesion - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- Procedural cohesion - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- Communicational cohesion - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- Sequential cohesion - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- Functional cohesion - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling
Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.
There are five levels of coupling, namely -

- Content coupling - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- Common coupling- When multiple modules have read and write access to some global data, it is called common or global coupling.
- Control coupling- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- Stamp coupling- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- Data coupling- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

5.4 SOFTWARE DESIGN STRATEGIES

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find

optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.
There are multiple variants of software design. Let us study them briefly:
Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.
There are multiple variants of software design. Let us study them briefly:

5.4.1 Structured Design

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.
Structured design is mostly based on 'divide and conquer' strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.
The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.
These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -
Cohesion - grouping of all functionally related elements.
Coupling - communication between different modules.
A good structured design has high cohesion and low coupling arrangements.

5.4.2 Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.
Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.
Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions change the data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

5.4.3 Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.
Let us see the important concepts of Object Oriented Design:

- Objects - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- Classes - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object. In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
- Encapsulation - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- Inheritance - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- Polymorphism - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Design Process
Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them are defined.
- Application framework is defined.

5.5 SOFTWARE ANALYSIS & DESIGN TOOLS

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.
Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.
Let us see few analysis and design tools used by software designers:

5.5.1 Data Flow Diagram

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

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

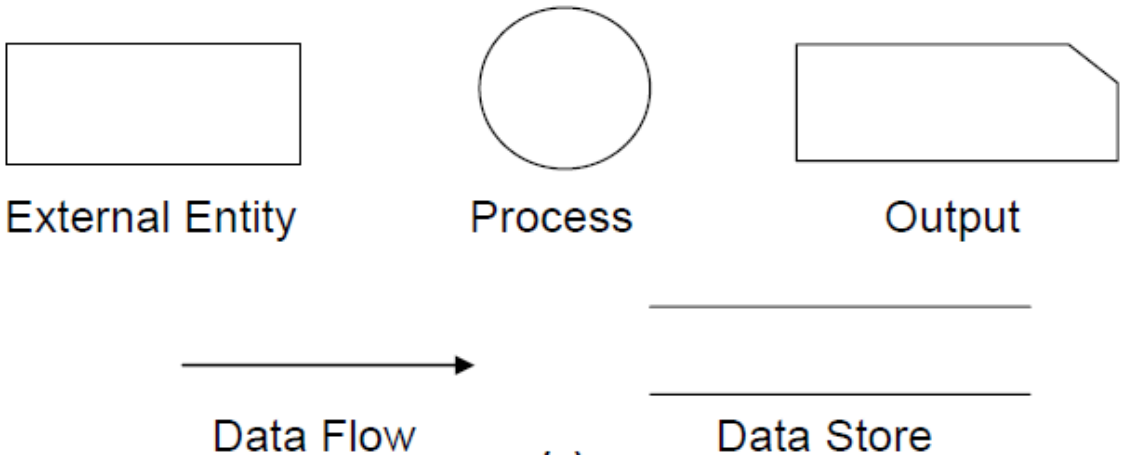
Types of DFD

Data Flow Diagrams are either Logical or Physical.

- Logical DFD - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.
- Physical DFD - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components

DFD can represent Source, destination, storage and flow of data using the following set of components -



- Entities - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.
- Process - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- Data Storage - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- Data Flow - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

5.5.2 Importance of DFDs in a good software design

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions.

In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

CHAPTER 6

6. CODING

Coding- The objective of the coding phase is to transform the design of a system into code in a high level language and then to unit test this code. The programmers adhere to standard and well defined style of coding which they call their coding standard. The main advantages of adhering to a standard style of coding are as follows:

- A coding standard gives uniform appearances to the code written by different engineers
- It facilitates code of understanding.
- Promotes good programming practices.

For implementing our design into a code, we require a good high level language. A programming language should have the following features:

6.1 Characteristics of a Programming Language

- Readability: A good high-level language will allow programs to be written in some ways that resemble a quite-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.

- Portability: High-level languages, being essentially machine independent, should be able to develop portable software.

- Generality: Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

- Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.

- Error checking: Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.

- Cost: The ultimate cost of a programming language is a function of many of its characteristics.

- Familiar notation: A language should have familiar notation, so it can be understood by most of the programmers.
- Quick translation: It should admit quick translation.
- Efficiency: It should permit the generation of efficient object code.
- Modularity: It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.
- Widely available: Language should be widely available and it should be possible to provide translators for all the major machines and for all the major operating systems.

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.

6.2 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.

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

2. 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.

3. 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.

4. 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.

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. Do not use goto statements: Use of goto statements makes a program unstructured and 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 Throughs

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 through, inspection, and verification are time-consuming.

6.3 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.

6.4 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 a software product. Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

6.5 Verification Vs 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.

6.6 BLACK-BOX 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.

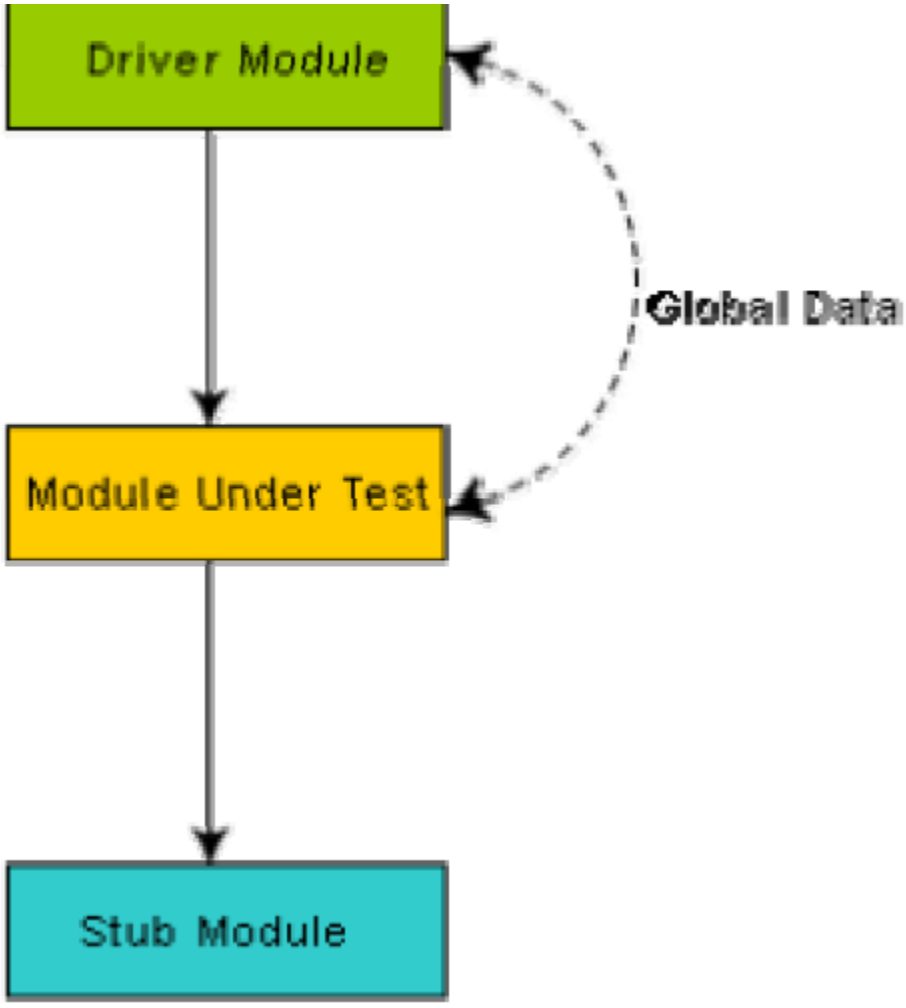
Unit Testing

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:

• The procedures 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 are 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. 19.1. 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 contains 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

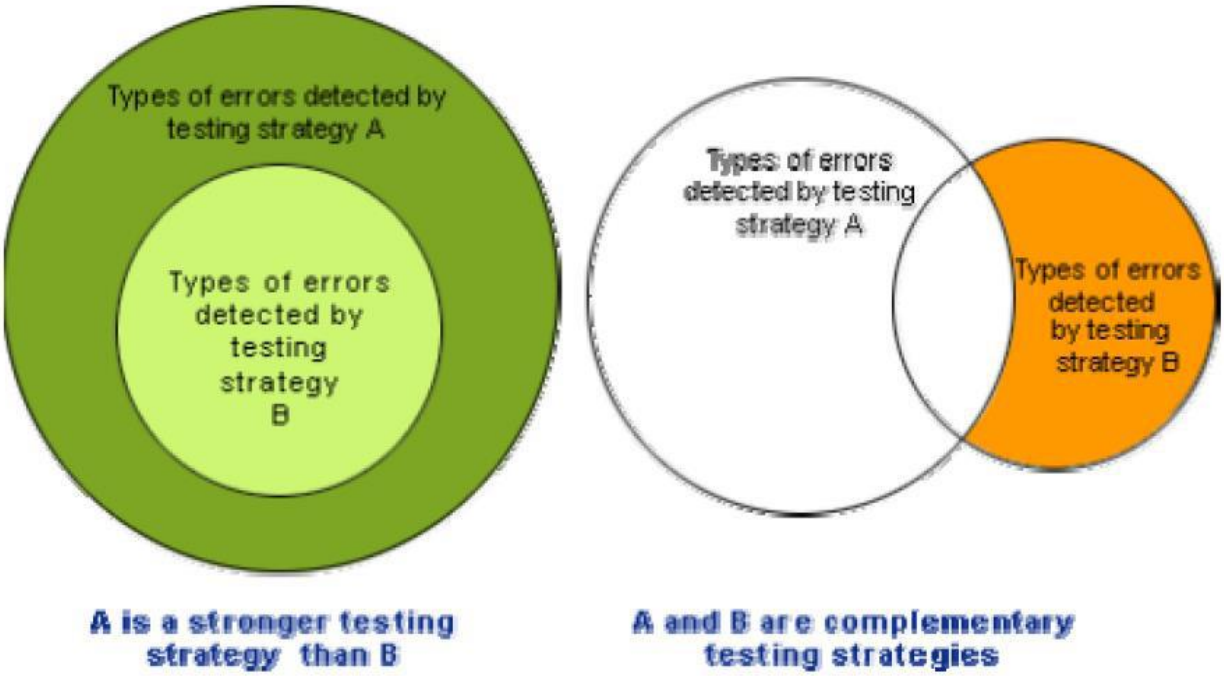
Equivalence Class Partitioning

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.

6.7 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.



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:

```
int compute_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.

Branch Coverage

In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program's control flow graph is traversed 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. 20.2). 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 these types of statements. Fig. 20.2 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. 20.3.

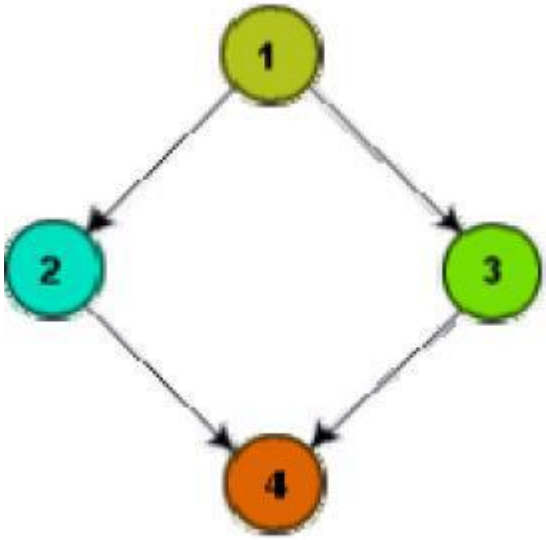
Sequence:

```
a=5;
b = a*2-1;
```

Selection:

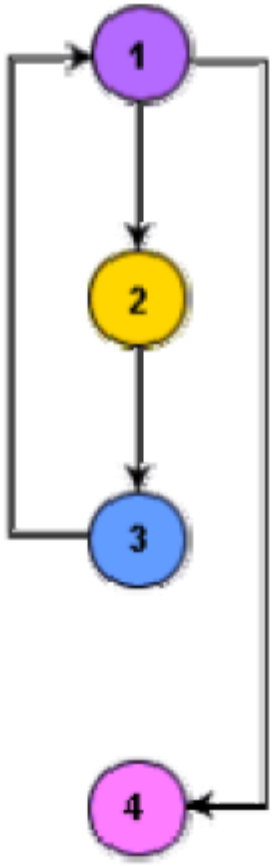
```
if (a>b)
c = 3;

else
c =5;
c=c*c;
```



Iteration :

```
while (a>b)
{
b=b -1;
b=b*a;
}
c = a+b;
```



6.8 DEBUGGING, INTEGRATION AND SYSTEM TESTING

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 those 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.

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 mistake 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 analysis methods. 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.

6.9 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
- Bottom- up approach
- Top-down 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:

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

o 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 test tests the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance test tests 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 multi programmed 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 = $N–n = n\times(S – s)/s$

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.

Regression Testing

Regression testing does not belong to either unit test, integration test, or system testing. Instead, it is a separate dimension to these three forms of testing. The functionality of regression testing has been discussed earlier.