# 1.Introduction to Software Engineering

## 1. Objectives

The objective of this lesson is to make the students acquainted with the introductory concepts of software engineering. To make them familiar with the problem of software crisis this has ultimately resulted into the development of software engineering. After studying this lesson, the students will:

1. Understand what is software crisis?
2. What are software engineering and its importance?

### 1.1 Introduction

In order to develop a software product, user needs and constraints must be determined and explicitly stated; the product must be designed to accommodate implementers, users and maintainers; the source code must be carefully implemented and thoroughly tested; and supporting documents must be prepared. Software maintenance tasks include analysis of change request, redesign and modification of the source code, thorough testing of the modified code, updating of documents to reflect the changes and the distribution of modified work products to the appropriate user. The need for systematic approaches to development and maintenance of software products became apparent in the 1960s. Many software developed at that time were subject to costOver runs, schedule slippage, lack of reliability, inefficiency, and lack of user acceptance. As computer systems become larger and complex, it became apparent that the demand for computer software was growing faster than our ability to produce and maintain it. As a result the field of software engineering has evolved into a technological discipline of considerable importance.

### 1.2 Software Engineering

### Software Engineering (SE) is the design, development, and documentation of software by applying technologies and practices from computer science, project management, engineering, application domains, interface design, digital asset management and other fields.

## 1.3 The Need for Software Engineering

Software is often found in products and situations where very high reliability is expected, even under demanding conditions, such as monitoring and controlling nuclear power plants, or keeping a modern airliner aloft Such applications contain millions of lines of code, making them comparable in complexity to the most complex modern machines. For example, a modern airliner has several million physical parts (and the space shuttle about ten million parts), while the software for such an airliner can run to 4 million lines of code.

## 1.5 Software Characteristics

The fundamental difference between a software and hardware is that software is a conceptual entity while hardware is physical entity. When the hardware is built, the process of building a hardware results in a physical entity, which can be easily measured. Software being a logical has the different characteristics that are to be understood.

➢**Software is developed or engineered but it is not manufactured in theClassical sense.**

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.

Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

# 2. Software Life Cycle Models

# 2.0 Objectives

The objective of this lesson is to introduce the students to the concepts ofSoftware life cycle models. After studying this lesson, they will:

* Understand a number of process models like waterfall model, spiral model, prototyping and iterative enhancement.
* Come to know the merits/demerits, and applicability of different models.

**2.1 Introduction**

A software process is a set of activities and associated results which lead to the production of a software product. There are many different software processes; there are fundamental activities which are common to all software processes. These are:

➢**Software specification:** The functionality of the software and constraints onits operations are defined.

## ➢Software design and implementation: The software to meet thespecification is produced.

**Software validation:** The software must be validated to ensure that it doeswhat the customer wants.

**Software evolution:** The software must evolve to meet changing customerneeds.

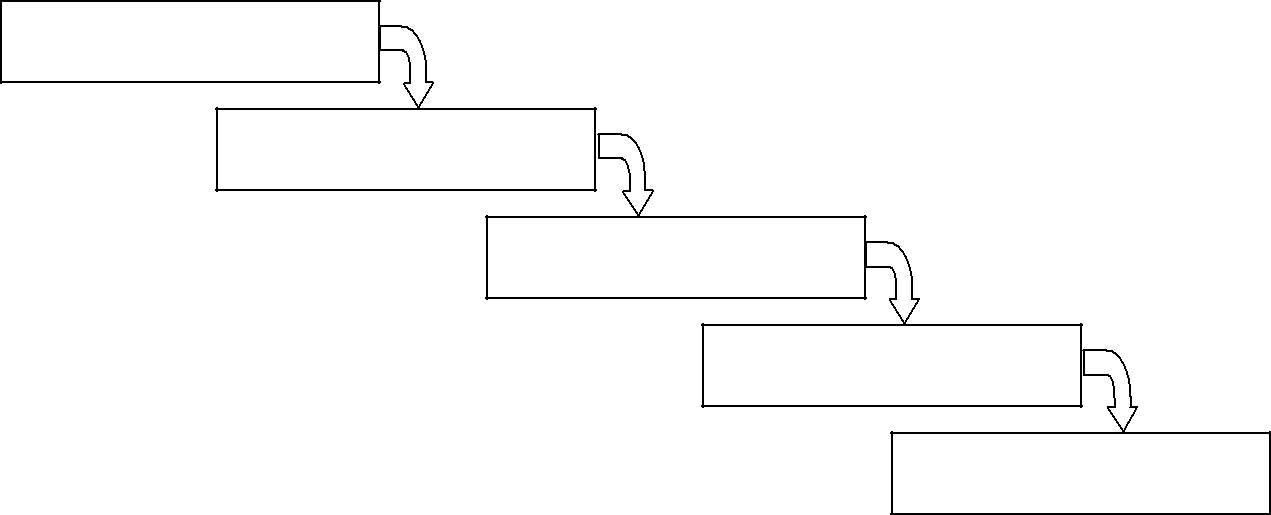
**2.2 Software process model**

Software process model is an abstract representation of a software process. A number of software models are below.

### 2.2.1 Water fall model

The waterfall model is a sequential software development model in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The origin of the term "waterfall" is often cited to be an article published in 1970 by W. W. Royce; ironically, Royce himself advocated an iterative approach to software development and did not even use the term "waterfall". Royce originally described what is now known as the waterfall model as an example of a method that he argued "is risky and invites failure".

## Usage of the waterfall model



REQUIREMENTS

DESIGN

IMPLEMENTATION

VERIFICATION

MAINTENANCE

**Figure Water Fall Model**

The unmodified "waterfall model". Progress flows from the top to the bottom, likea waterfall. In Royce's original waterfall model, the following phases are followed perfectly in order:

1. Requirements specification
2. Design
3. Construction (implementation or coding)
4. Integration
5. Testing and debugging (verification)
6. Installation
7. Maintenance

**2.2.2 Software Prototyping and Requirements Engineering**

The conventional waterfall software life cycle model (or software process) is used to characterize the phased approach for software development and maintenance. Software life cycle phase names differ from organization to organization. The software process includes the following phases:

1. requirements formulation and analysis, specification
2. design
3. coding
4. testing
5. Maintenance

Software customers find it very difficult to express their requirements. Careful requirement analysis with systematic review help to reduce the uncertainty about what the system should do. However there is no substitute for trying out a requirement before agreeing to it. This is possible if the prototype is available. A software prototype supports two requirement engineering process activities:

1. Requirement elicitation: System prototype helps the user in identifying his requirements better. He can experiment with it to see how the system supports their work. In this process they can get new ideas and find strength and weakness in the software.
2. Requirement validation: The prototype may reveal errors and omissions in the requirements which have been proposed.

A significant risk in software development is requirement errors and omissions and prototyping help in risk analysis and reduction. So prototyping is part of requirement engineering process. But now prototyping has been introduced throughout the conventional, waterfall software life cycle model. Now a day many systems are developed using an evolutionary approach where an initial version is created quickly and modified to produce a final system.

Two forms of life cycle models, Throwaway prototyping and evolutionary prototyping, have emerged around prototyping technology.

## Prototyping Pitfalls

Prototyping has not been as successful as anticipated in some organizations for a variety of reasons. Training, efficiency, applicability, and behaviour can each have a negative impact on using software prototyping techniques.

## Learning Curve

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

## Tool Efficiency

Prototyping techniques outside the domain of conventional programming languages can have execution inefficiencies with the associated tools. The efficiency question was argued as a negative aspect of prototyping.

## Applicability

Application domain has an impact on selecting a prototyping technique. There would be limited benefit to using a technique not supporting real-time features in a process control system. The control room user interface could be described, but not integrated with sensor monitoring deadlines under this approach.

## Undefined Role Models for Personnel

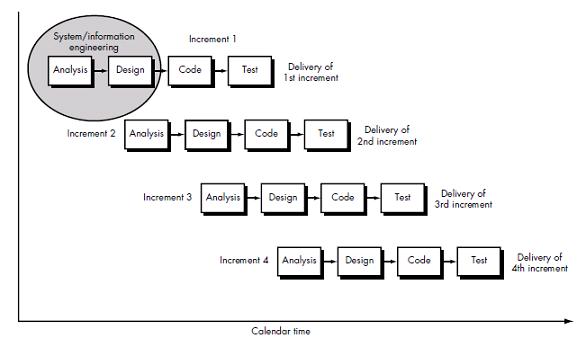
This new approach of providing feedback early to the end user has resulted in a problem related to the behaviour of the end user and developers. An end user with a previously unfortunate system development effort can be biased in future interactions with development teams.

### Iterative Enhancement

The iterative enhancement model counters the third limitation of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the water- fall model. Furthermore, as in prototyping, the increments provide feedback to the client that is useful for determining the final requirements of the system.

In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspectsof the problem that are easy to understand and implement and which form a useful and usable system. A project control list is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far the project is at any given step from the final system.

Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called the design phase, implementation phase, and analysis phase. The process is iterated until the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement process model is shown in Figure :



## Figure the Iterative Enhancement Model

The project control list guides the iteration steps and keeps track of all the tasks that must be done. Based on the analysis, one of the tasks in the list can include redesign, of defective components or redesign of the entire system. However, redesign of the system will generally occur only in the initial steps. In the later steps, the design would have stabilized and there is less chance of redesign.

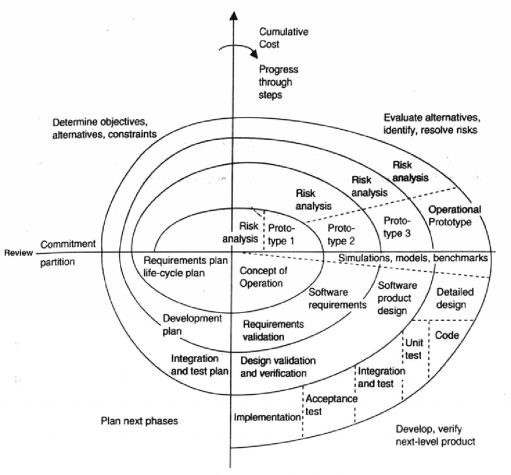
Each entry in the list is a task that should be performed in one step of the iterative enhancement process and should be simple enough to be completely understood. Selecting tasks in this manner will minimize the chances of error and reduce the redesign work. The design and implementation phases of each step can be performed in a top-down manner or by using some other technique.

One effective use of this type of model is product development, in which the developers themselves provide the specifications and, therefore, have a lot of control on what specifications go in the system and what stay out. In fact, most products undergo this type of development process. First, a version is released that contains some capability. Based on the feedback from users and experience with this version, a list of additional features and capabilities is generated. These features form the basis of enhancement of the software, and are included in the next version. In other words, the first version contains some core capability and then more features are added to later versions.

However, in a customized software development, where the client has to essentially provide and approve the specifications, it is not always clear how this process can be applied. Another practical problem with this type of development project comes in generating the business contract-how will the cost of additional features be determined and negotiated, particularly because the client organization is likely to be tied to the original vendor who developed the first version. Overall, in these types of projects, this process model can be useful if the "core" of the application to be developed is well understood and the “increments" can be easily defined and negotiated. In client-oriented projects, this process has the major advantage that the client's organization does not have to pay for the entire software together; it can get the main part of the software developed and, perform cost-benefit analysis for it before enhancing the software with more capabilities.

**2.2.3. The Spiral Model**

This model was originally proposed by Bohem (1988). As it is clear from the name, the activities in this model can be organized like a spiral that has many cycles. The radial dimension represents the cumulative cost incurred in accomplishing the steps done so far, and the angular dimension represents the progress made in completing each cycle of the spiral. The model is shown in



## Figure Boehm’s spiral model of the software process

Each cycle in the spiral is split into four sectors:

1. **Objective setting:** Each cycle in the spiral begins with the identification ofobjectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist. This is the first quadrant of the cycle (upper-left quadrant).
2. **Risk Assessment and reduction:** The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project. Risks reflect the chances that some of the objectives of the project may not be met.
3. **Development and validation:** The next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping.
4. **Planning:** Next, the software is developed, keeping in mind the risks. Finally,the next stage is planned. The project is reviewed and a decision made whether to continue with a further cycle of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The development step depends on the remaining risks. For example, if performance or user-interface risks are considered more important than the program development risks, the next step may be an evolutionary development that involves developing a more detailed prototype for resolving the risks.

**2.2.4 Agile Development Models**

In earlier days Iterative Waterfall model was very popular to complete a project. But nowadays developers face various problems while using it to develop a software. The main difficulties included handling change requests from customers during project development and the high cost and time required to incorporate these changes. To overcome these drawbacks of Waterfall model, in the mid-1990s the Agile Software Development model was proposed.

Agile SDLC models are given below:

1. Scrum
2. Extreme programming (XP)

Agile model is the combination of iterative and incremental process models. Steps involve in agile SDLC models are:

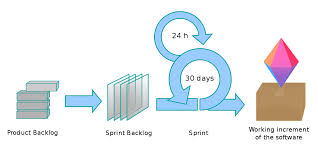
1. Requirement gathering
2. Requirement Analysis
3. Design
4. Coding
5. Unit testing
6. Acceptance testing

**Advantages:**

1. Working through Pair programming produce well written compact programs which has fewer errors as compared to programmers working alone.
2. It reduces total development time of the whole project.
3. Customer representative get the idea of updated software products after each iretation. So, it is easy for him to change any requirement if needed.

**Disadvantages:**

1. Due to lack of formal documents, it creates confusion and important decisions taken during different phases can be misinterpreted at any time by different team members.
2. Due to absence of proper documentation, when the project completes and the developers are assigned to another project, maintenance of the developed project can become a problem.



# Project Planning

## 3.0 Objectives

Project planning is an important issue in the successful completion of a software project. The objective of this lesson is to make the students familiar with the factors affecting the cost of the software, different versions of COCOMO and the problems and criteria to evaluate the models.

## 3.1 Introduction

Software cost estimation is the process of predicting the amount of effort required to build a software system. Software cost estimation is one of the most difficult and error prone task in software engineering. Cost estimates are needed throughout the software lifecycle. Preliminary estimates are required to determine the feasibility of a project. Detailed estimates are needed to assist with project planning. The actual effort for individual tasks is compared with estimated and planned values, enabling project managers to reallocate resources when necessary.

Analysis of historical project data indicates that cost trends can be correlated with certain measurable parameters. This observation has resulted in a wide range of models that can be used to assess, predict, and control software costs on a realtime basis. Models provide one or more mathematical algorithms that compute

cost as a function of a number of variables.

## 3.2 Cost factor

There are a number of factors affecting the cost of the software. The major one are listed below:

1. **Programmer ability:** Results of the experiments conducted by Sackmanshow a significant difference in individual performance among the programmers. The difference between best and worst performance were factors of 6 to I in program size, 8 to 1 in execution time, 9 to 1 in development time, 18 to 1 in coding time, and 28 to 1 in debugging time.
2. **Product Complexity:** There are generally three acknowledged category ofthe software: application programs, utility programs and system programs. According to Brook utility programs are three times as difficult to write as application programs, and that system programs are three times as difficult to write as utility programs. So it is a major factor influencing the cost of software.
3. **Product Size:** It is obvious that a large software product will be moreexpensive than a smaller one.
4. **Available time:** It is generally agreed that software projects require more total efforts if development time is compressed or expanded from the optimal time.
5. **Required reliability:** Software reliability can be defined as the probability that a program will perform a required function under stated conditions for a stated period of time. Reliability can be improved in a software, but there is a cost associated with the increased level of analysis, design, implementation, verification and validation efforts that must be exerted to ensure high reliability.
6. **Level of technology:** The level of technology is reflected by the programming language, abstract machine, programming practices and software tools used. Using a high level language instead of assembly language will certainly improve the productivity of programmer thus resulting into a decrease in the cost of software.

## 3.3 COCOMO

Boehm's COCOMO model is one of the mostly used models commercially. The first version of the model delivered in 1981 and COCOMO II is available now. COCOMO is a model designed by Barry Boehm to give an estimate of the number of man-months it will take to develop a software product. This “ConstructiveCostModel" is based on a study of about sixty projects at TRW, a Californian automotive and IT company, acquired by Northrop Grumman in late2002. The programs examined ranged in size from 2000 to 100,000 lines of code, and programming languages used ranged from assembly to PL/I. COCOMO consists of a hierarchy of three increasingly detailed and accurate forms.

1. Basic COCOMO - is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code.
2. Intermediate COCOMO - computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes.
3. Detailed COCOMO - incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

### 3.3.1 Basic COCOMO

Basic COCOMO is a form of the COCOMO model. COCOMO may be applied to three classes of software projects. These give a general impression of the

Software project.

* **Organic projects** – These are relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements.
* **Semi-detached projects** – These are intermediate (in size and

Complexity) software projects in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements.

* **Embedded projects** – These are software projects that must be developedwithin a set of tight hardware, software, and operational constraints.

The basic COCOMO equations take the form

E=a (KLOC) b

D=c (E) d

P=E/D

Where E is the effort applied in person-months, D is the development time in chronological months, KLOC is the estimated number of delivered lines of code for the project (expressed in thousands), and P is the number of people required. The coefficients ab, bb, cb and db are given in the table 4.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Software project | a | b | c | D |
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

## Table Coefficients for Basic COCOMO

Basic COCOMO is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is necessarily limited because of its lack of factors to account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and other project attributes

Known to have a significant influence on software costs.

### 3.3.2 Intermediate COCOMO

The Intermediate COCOMO is an extension of the Basic COCOMO model, and is used to estimate the programmer time to develop a software product. This extension considers a set of "cost driver attributes" that can be grouped into four major categories, each with a number of subcategories:

### 3.3.3 Detailed COCOMO

The Advanced COCOMO model computes effort as a function of program size and a set of cost drivers weighted according to each phase of the software lifecycle. The Advanced model applies the Intermediate model at the component level, and then a phase-based approach is used to consolidate the estimate.

The 4 phases used in the detailed COCOMO model are: requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT), and integration and test (IT). Each cost driver is broken down by phase as in the example shown in Table 4.5.

Estimates made for each module are combined into subsystems and eventually an overall project estimate. Using the detailed cost drivers, an estimate is determined for each phase of the lifecycle.

## Advantages of COCOMO

1. COCOMO is transparent; you can see how it works unlike other models such as SLIM.
2. Drivers are particularly helpful to the estimator to understand the impact of different factors that affect project costs.

## Drawbacks of COCOMO

1. It is hard to accurately estimate KDSI early on in the project, when most effort estimates are required.
2. KDSI, actually, is not a size measure it is a length measure.
3. Extremely vulnerable to mis-classification of the development mode
4. Success depends largely on tuning the model to the needs of the
5. Organization, using historical data which is not always available

# Software Requirement Analysis &Specification

## 4.1 Introduction

The analysis phase of software development is concerned with project planning and software requirement definition. To identify the requirements of the user is a tedious job. The description of the services and constraints are the requirements for the system and the process of finding out, analyzing, documenting, and checking these services is called requirement engineering. The goal ofrequirement definition is to completely and consistently specify the requirements for the software product in a concise and unambiguous manner, using formal notations as appropriate. The software requirement specification is based on the system definition. The requirement specification will state the “what of” the software product without implying “how”. Software design is concerned with specifying how the product will provide the required features.

## 4.2. Software system requirements

Software system requirements are classified as functional requirements and non-functional requirements.

### 4.2.1. Functional requirements

The functional requirements for a system describe the functionalities or services that the system is expected to provide. They provide how the system should react to particular inputs and how the system should behave in a particular

situation.

**4.2.2 Non-functional requirements t**hese are constraints on the services or functionalities offered by the system.

They include timing constraints, constraints on the development process, standards etc. These requirements are not directly concerned with the specific function delivered by the system. They may relate to such system properties such as reliability, response time, and storage. They may define the constraints on the system such as capabilities of I/O devices and the data representations used in system interfaces.

## 4.3 Software requirement specification

It is the official document of what is required of the system developers. It consists of user requirements and detailed specification of the system requirements. According to Henninger there are six requirements that an SRS should satisfy:

1. It should specify only external system behaviour.
2. It should specify constraints on the implementation.
3. It should be easy to change.
4. It should serve as a reference tool for system maintainers.
5. It should record forethought about the life cycle of the system.
6. It should characterize acceptable response to undesired events.

#### 4.4 Characteristics of SRS

The desirable characteristics of an SRS are following:

1. **Correct:** An SRS is correct if every requirement included in the SRSrepresents something required in the final system.
2. **Complete:** An SRS is complete if everything software is supposed to do andthe responses of the software to all classes of input data are specified in the SRS.
3. **Unambiguous:** An SRS is unambiguous if and only if every requirementstated has one and only one interpretation.
4. **Verifiable:** An SRS is verifiable if and only if every specified requirement isverifiable i.e. there exists a procedure to check that final software meets the requirement.
5. **Consistent:** An SRS is consistent if there is no requirement that conflicts withanother.
6. **Traceable:** An SRS is traceable if each requirement in it must be uniquelyidentified to a source.
7. **Modifiable:** An SRS is modifiable if its structure and style are such that anynecessary change can be made easily while preserving completeness and consistency.
8. **Ranked:** An SRS is ranked for importance and/or stability if for eachrequirement the importance and the stability of the requirements are indicated.

#### 4.5 Components of an SRS

An SRS should have the following components:

1. Functionality
2. Performance
3. Design constraints
4. External Interfaces

### Functionality

Here functional requirements are to be specified. It should specify which outputs should be produced from the given input. For each functional requirement, a detailed description of all the inputs, their sources, range of valid inputs, the units of measure are to be specified. All the operation to be performed on input should

also be specified.

### Performance requirements

In this component of SRS all the performance constraints on the system should be specified such as response time, throughput constraints, number of terminals to be supported, number of simultaneous users to be supported etc.

### Design constraints

Here design constraints such as standard compliance, hardware limitations, Reliability, and security should be specified. There may be a requirement that system will have to use some existing hardware, limited primary and/or secondary memory. So it is a constraint on the designer. There may be some standards of the organization that should be obeyed such as the format of reports. Security requirements may be particularly significant in defense systems. It imposes a restriction sometimes on the use of some commands, control access to data; require the use of passwords and cryptography techniques etc.

### External Interface requirements

Software has to interact with people, hardware, and other software. All these interfaces should be specified. User interface has become a very important issue now a day. So the characteristics of user interface should be precisely specified and should be verifiable.

# Software Design and Implementation

## 5.1 Introduction

Design is an iterative process of transforming the requirements specification into a design specification. Consider an example where Mrs. & Mr. XYZ want a new house. Their requirements include,

1. a room for two children to play and sleep
2. a room for Mrs. & Mr. XYZ to sleep
3. a room for cooking
4. a room for dining
5. a room for general activities

and so on. An architect takes these requirements and designs a house. The architectural design specifies a particular solution. In fact, the architect may produce several designs to meet this requirement. For example, one maymaximize children’s room, and other minimizes it to have large living room. In addition, the style of the proposed houses may differ: traditional, modern and two-storied. All of the proposed designs solve the problem, and there may not be a “best” design.Software design can be viewed in the same way. We use requirements specification to define the problem and transform this to a solution that satisfies all the requirements in the specification. Design is the first step in the development phase for any engineered product. The designer goal is to produce a model of an entity that will later be built.

## 5.2 Definitions for Design

1. “The process of defining the architecture, component, interfaces and other characteristics of a system or component”
2. The process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization.

Without Design, System will be

1. Unmanageable since there is no concrete output until coding. Therefore it is difficult to monitor & control.
2. Inflexible since planning for long term changes was not given due emphasis.
3. UN maintainable since standards & guidelines for design & construction are not used. No reusability consideration. Poor design may result in tightly coupled modules with low cohesion. Data disintegrated may also result.
4. Inefficient due to possible data redundancy and unturned code.
5. Not portable to various hardware / software platforms.

Design is different from programming. Design brings out a representation for the program – not the program or any component of it. The difference is tabulated below.

## 5.3 Qualities of a Good Design

**Functional:** It is a very basic quality attribute. Any design solution should work,and should be construct able.

**Efficiency:** This can be measured through

1. run time (time taken to undertake whole of processing task or transaction)
2. response time (time taken to respond to a request for information)
3. throughput (no. of transactions / unit time)
4. memory usage, size of executable, size of source, etc

**Flexibility:** It is another basic and important attribute. The very purpose of doingdesign activities is to build systems that are modifiable in the event of any changes in the requirements.

**Portability & Security:** These are to be addressed during design - so that suchneeds are not “hard-coded” later.

**Reliability:** It tells the goodness of the design - how it work successfully (Moreimportant for real-time and mission critical and on-line systems).

**Economy:** This can be achieved by identifying re-usable components.

**Usability:** Usability is in terms of how the interfaces are designed (clarity,aesthetics, directness, forgiveness, user control, ergonomics, etc) and howmuch time it takes to master the system.

## 5.4 Modularity

There are many definitions of the term "module." They range from "a module is a FORTRAN subroutine" to "a module is an Ada package" to "a module is a work assignment for an individual programmer". All of these definitions are correct, in the sense that modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved. Modular systems consist of well-defined, manageable units with well-defined interfaces among the units.

Desirable properties of a modular system include:

1. Each processing abstraction is a well-defined subsystem that is potentially useful in other applications.
2. Each function in each abstraction has a single, well-defined purpose.
3. Each function manipulates no more than one major data structure.
4. Functions share global data selectively. It is easy to identify all routines that share a major data structure.
5. Functions that manipulate instances of abstract data types are encapsulatedwith the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

### 5.5 Coupling and cohesion

A fundamental goal of software design is to structure the software product so that the number and complexity of interconnection between modules is minimized. A good heuristic for achieving this goal involves the concepts of

coupling and cohesion.

### Coupling

Coupling is the measure of strength of association established by a connection from one module to another. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system (‘ripple effect’). The use of global variables can result in an enormous number of connections between the modules of a program. The degree of coupling between two modules is a function of several factors: (1) How complicated the connection is, (2) Whether the connection refers to the module itself or something inside it, and (3) What is being sent or received. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Low coupling means that one module does not have to be concerned with the internal implementation of another module, and interacts with another module with a stable interface. With low coupling, a change in one module will not require a change in the implementation of another module. Low coupling is a sign of a well structured computer system.

However, in order to achieve maximum efficiency, a highly coupled system is sometimes needed. In modern computing systems, performance is often traded for lower coupling; the gains in the software development process are greater than the value of the running performance gain.

Low-coupling / high-cohesion is a general goal to achieve when structuring computer programs, so that they are easier to understand and maintain. The concepts are usually related: low coupling implies high cohesion and vice versa. In the field of object-oriented programming, the connection between classes tends to get lower (low coupling), if we group related methods of a class together (high cohesion).

### Data Coupling

Two modules are data coupled if they communicate by parameters (each being an elementary piece of data).E.g. sin (theta) returning sine value, calculate interest (amount, interest rate, term) returning interest amt.

### Stamp Coupling (Data-structured coupling)

Two modules are stamp coupled if one passes to other a composite piece of data

(a piece of data with meaningful internal structure). Stamp coupling is when modules share a composite data structure, each module not knowing which part of the data structure will be used by the other (e.g. passing a student record to a

function which calculates the student's GPA)

### Control Coupling

Two modules are control coupled if one passes to other a piece of information intended to control the internal logic of the other. In Control coupling, one modulecontrols logic of another, by passing it information on what to do (e.g. passing a what-to-do flag).

### External coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

### Common coupling

Two modules are common coupled if they refer to the same global data area.

Instead of communicating through parameters, two modules use a global data

### Content coupling

Two modules exhibit content coupled if one refers to the inside of the other in any way (if one module ‘jumps’ inside another module). E.g. Jumping inside a module violate all the design principles like abstraction, information hiding and modularity.

In object-oriented programming, subclass coupling describes a special type of coupling between a parent class and its child. The parent has no connection to the child class, so the connection is one way (i.e. the parent is a sensible class on its own). The coupling is hard to classify as low or high; it can depend on the situation.

**Coupling is increased between two classes A and B if:**

1. A has an attribute that refers to (is of type) B.
2. A calls on services of a B object.
3. A has a method which references B (via return type or parameter).
4. A is a subclass of (or implements) B.

**Disadvantages of high coupling include:**

1. A change in one class forces a ripple of changes in other classes.
2. Difficult to understand a class in isolation.
3. Difficult to reuse or test a class because dependent class must also be included.

One measure to achieve low coupling is functional design: it limits the responsibilities of modules. Modules with single responsibilities usually need to communicate less with other modules, and this has the virtuous side-effect of reducing coupling and increasing cohesion in many cases.

### Cohesion

Designers should aim for loosely coupled and highly cohesive modules. Coupling is reduced when the relationships among elements not in the same module are minimized. Cohesion on the other hand aims to maximize the relationships among elements in the same module. Cohesion is a good measure of the maintainability of a module. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understand ability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand.

The types of cohesion, in order of lowest to highest, are as follows:

1. Coincidental Cohesion (Worst)
2. Logical Cohesion
3. Temporal Cohesion
4. Procedural Cohesion
5. Communicational Cohesion
6. Sequential Cohesion
7. Functional Cohesion (Best)

### Coincidental cohesion (worst)

Coincidental cohesion is when parts of a module are grouped arbitrarily; the parts have no significant relationship (e.g. a module of frequently used functions).

### Logical cohesion

Logical cohesion is when parts of a module are grouped because of a slight relation (e.g. using control coupling to decide which part of a module to use, such as how to operate on a bank account).

### Temporal cohesion

In a temporally bound (cohesion) module, the elements are related in time. Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

### Procedural cohesion

Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

### Communicational cohesion

Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a method updateStudentRecord which operates on a student record, but the actions which the method performs are not clear).

### Sequential cohesion

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part (e.g. a function which reads data from a file and processes the data).

### Functional cohesion (best)

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (a perfect module).Since cohesion is a ranking type of scale, the ranks do not indicate a steady progression of improved cohesion. Studies by various people including Larry

Constantine and Edward Yourdon as well as others indicate that the first two types of cohesion are much inferior to the others and that module with communicational cohesion or better tend to be much superior to lower types of cohesion. The seventh type, functional cohesion, is considered the best type.However, while functional cohesion is considered the most desirable type of cohesion for a software module, it may not actually be achievable. There are many cases where communicational cohesion is about the best that can be attained in the circumstances. However the emphasis of a software design should be to maintain module cohesion of communicational or better since these types of cohesion are associated with modules of lower lines of code per module with the source code focused on a particular functional objective with less extraneous or unnecessary functionality, and tend to be reusable under a greater variety of conditions.

##### 5.6 Structured Programming

The goal of structured programming is to linearism control flow through a computer program so that the execution sequence follows the sequence in which the code is written. The dynamic structure of the program than resemble the static structure of the program. This enhances the readability, testability, and modifiability of the program. This linear flow of control can be achieved by restricting the set of allowed program construct to single entry, single exit formats. These issues are discussed in the following section:

# 5.7 Coding Objectives

The objective of this lesson is to make the students familiar

1. With the concept of coding.
2. Programming Style
3. Verification and validations techniques.

## 5.7.1 Introduction

The coding is concerned with translating design specifications into source code. The good programming should ensure the ease of debugging, testing and modification. This is achieved by making the source code as clear and straightforward as possible. An old saying is “Simple is great”. Simplicity, clarity and elegance are the hallmarks of good programs. Obscurity, cleverness, and complexity are indications of inadequate design. Source code clarity is enhanced by structured coding techniques, by good coding style, by appropriate supporting documents, by good internal comments etc. Production of high quality software requires that the programming team should have a thorough understanding of duties and responsibilities and should be provided with a well defined set of requirements, an architectural design specification, and a detailed design description.

## 5.8 Programming style

Programming style refers to the style used in writing the source code for a computer program. Most programming styles are designed to help programmers quickly read and understands the program as well as avoid making errors. (Older programming styles also focused on conserving screen space.) A good coding style can overcome the many deficiencies of a primitive programming language, while poor style can defeat the intent of en excellent language. The goal of good programming style is to provide understandable, straightforward, elegant code.The programming style used in a particular program may be derived from the coding standards or code conventions of a company or other computing organization, as well as the preferences of the actual programmer. Programming styles are often designed for a specific programming language (or language family) and are not used in whole for other languages. (Style considered good in C source code may not be appropriate for BASIC source code, and so on.) Good style, being a subjective matter, is difficult to concretely categorize; however, there are several elements common to a large number of programming styles.

Programming styles are often designed for a specific programming language and are not used in whole for other languages. So there is no single set of rules that can be applied in every situation; however there are general guidelines that are widely applicable. These are listed below:

## 5.8.1Dos of good programming style

1. Use a few standards, agreed upon control constructs.
2. Use GOTO in a disciplined way.
3. Use user-defined data types to model entities in the problem domain.
4. Hide data structure behind access functions
5. Isolate machine dependencies in a few routines.
6. Use appropriate variable names
7. Use indentation, parentheses, blank spaces, and blank lines to enhance readability.

### 5.8.2 Don’ts of good programming style

1. Don’t be too clever.
2. Avoid null Then statement
3. Avoid Then If statement
4. Don’t nest too deeply.
5. Don’t use an identifier for multiple purposes.
6. Examine routines having more than five formal parameters.

### 6.Software Testing

* 1. **Introduction**

Software testing is the process used to help identify the correctness, completeness, security, andquality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders.

### 6.2 Software Verification and Validation Concepts and Definitions

Software Verification and Validation (V&V) is the process of ensuring that software being developed or changed will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification). The differences between verification and validation (shown in table 8.1) are unimportant except to the theorist;practitioners’ use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required.V&V is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the development process to ensure software requirements are complete and testable and that design, code, documentation, and data satisfy those requirements.

### Table Difference between verification and validation

|  |  |
| --- | --- |
| **Validation** | **Verification** |
| Am I building the right product? | Am I building the product right? |
| Determining if the system complieswith the requirements and performsfunctions for which it is intended andmeets the organization’s goals anduser needs. It is traditional and isperformed at the end of the project. | The review of interim work stepsandinterim deliverablesduringaprojct toensure they are acceptable.To determine  if the system isconsistent,adheres tostandards, usesreliable techniques andPrudentpractices, andperformstheselected functions in the correct manner. |
| Am I accessing the right data (in terms of the data required to satisfythe requirement)? | Am I accessing the data right (in the rightplace; in the right way)? |
| High level activity | Low level activity |
| Performed after a work product isproduced againstestablishedcriteria ensuring that the productintegratescorrectly into theenvironment | Performed during development on keyartifacts, like walkthroughs, reviews andinspections, mentor feedback, training, checklists and standards |
| Determination of correctness of thefinal software product by adevelopment project with respect to  the user needs and requirements. | Demonstrationofconsistency,completeness, and correctness of thesoftware at each stage and between eachstage of the development life cycle. |

#### 6.3. Testing Objectives

Glen Myres states a number of rules that can serves as testing objectives:

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

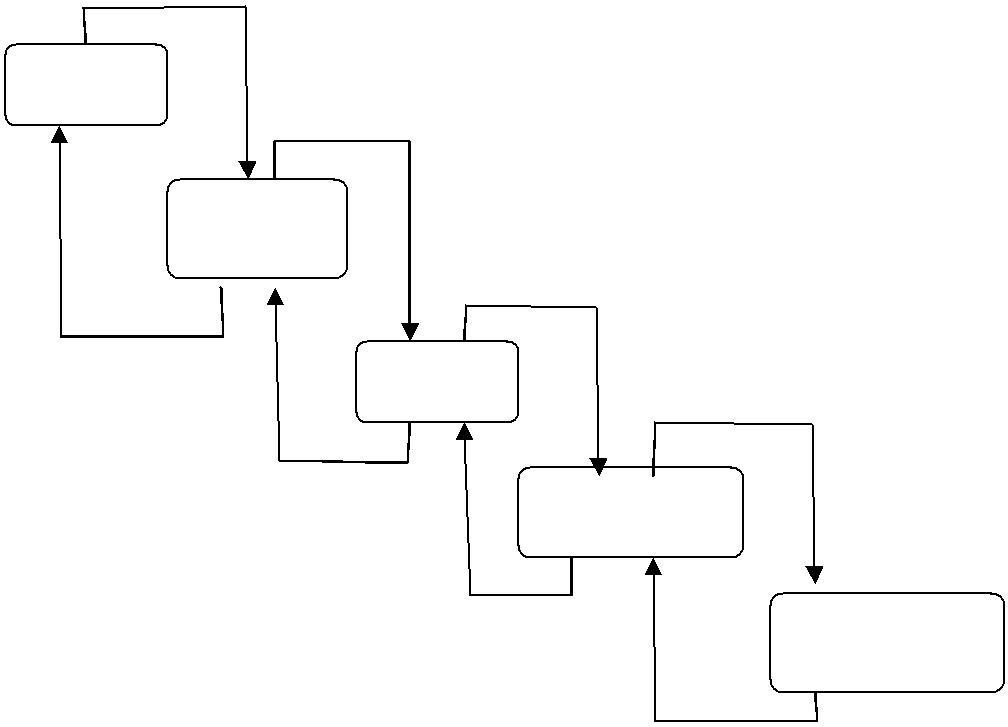
#### 6.4 Testing Principles

Davis suggested the following testing principles:

1. All tests should be traceable to customer requirements.
2. Tests should be planned long before testing begins.
3. The Pareto principle applies to software testing. According to this principle 80 percent of all errors uncovered during testing will likely to be traceable to 20 percent of all program modules. The problem is to isolate these 20 percent modules and test them thoroughly.
4. Testing should begin “in the small” and progress toward testing “in the large”.
5. Exhaustive testing is not possible.
6. To be most effective, testing should be conducted by an independent third party.

#### 6.5 Test Levels

1. Unit testing: It tests the minimal software item that can be tested. Each component is tested independently.
2. Module testing: A module is a collection of dependent components. So it is component integration testing and it exposes defects in the interfaces and interaction between integrated components.
3. Sub-system testing: It involves testing collection of modules which have been integrated into sub-systems. The sub-system test should concentrate on the detection of interface errors.
4. System testing: System testing tests an integrated system to verify that it meets its requirements. It is concerned with validating that the system meets its functional and non-functional requirements.
5. Acceptance testing: Acceptance testing allows the end-user or customer to decide whether or not to accept the product.



UNIT

TESTING

MODULE

TESTING

SYSTEM

TESTING

SUB

-

SYSTEM

TESTING

ACCEPTANCE

TESTING

### Figure Test levels

#### 6.6 SYSTEM TESTING

System testing involves two kinds of activities: integration testing and acceptance testing. Strategies for integrating software components into a functioning product include the bottom-up strategy, the top-down strategy, and the sandwichstrategy. Careful planning and scheduling are required to ensure that modules will be available for integration into the evolving software product when needed.The integration strategy dictates the order in which modules must be available, and thus exerts a strong influence on the order in which modules are written, debugged, and unit tested.

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests to verify that the implemented system satisfies its requirements. Acceptance tests are typically performed by the quality assurance and/or customer organizations. Depending on local circumstances, the development group may or may not be involved in acceptance testing. Integration testing and acceptance testing are in the following sections.

##### 6.6.1 Integration Testing

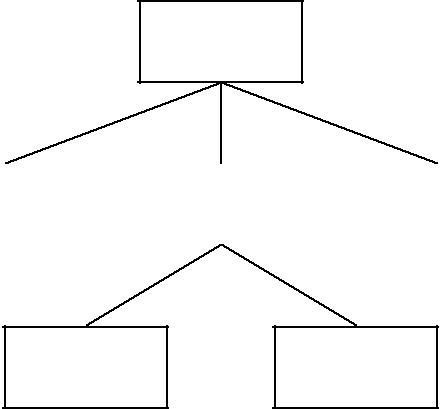
Three are two important variants of integration testing, (a) Bottom-up integration and top-down integration, which are the following sections:

###### 6.6.1.1 Bottom-up integration

Bottom-up integration is the traditional strategy used to integrate the components of a software system into a functioning whole. Bottom-up integration consists of unit testing, followed by subsystem testing, followed by testing of the entire system. Unit testing has the goal of discovering errors in the individual modules of the system. Modules are tested in isolation from one another in an artificial environment known as a "test harness," which consists of the driver programs and data necessary to exercise the modules. Unit testing should be asexhaustive as possible to ensure that each representative case handled by each module has been tested. Unit testing is eased by a system structure that is composed of small, loosely coupled modules. A subsystem consists of several modules that communicate with each other through well-defined interfaces.

###### 6.6.1.2Top-down integration

Top-down integration starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level "skeleton" has been thoroughly tested, it becomes the test harness for its immediately subordinate routines. Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.



MAIN

GET

PROC

PUT

SUB1

SUB2

### Figure

1. Test MAIN module, stubs for GET, PROC, and PUT are required.
2. Integrate GET module and now test MAIN and GET
3. Integrate PROC, stubs for SUBI, SUB2 are required.
4. Integrate PUT, Test MAIN, GET, PROC, PUT
5. Integrate SUB1 and test MAIN, GET, PROC, PUT, SUBI
6. Integrate SUB2 and test MAIN, GET, PROC, PUT, SUBI, SUB2

**Top-down integration offers several advantages:**

1. System integration is distributed throughout the implementation phase. Modules are integrated as they are developed.
2. Top-level interfaces are tested first and most often.
3. The top-level routines provide a natural test harness for lower-Level routines.
4. Errors are localized to the new modules and interfaces that are being added.

#### 6.6.2 Regression testing

Regression testing is an integral part of the extreme programming software development methodology. In this methodology, design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage in the software development cycle.

## Uses of regression testing

Regression testing can be used not only for testing the correctness of a program, but it is also often used to track the quality of its output. For instance in the design of a compiler, regression testing should track the code size, simulation time, and compilation time of the test suites.

### 6.6.3 Recovery testing

Many systems must recover from faults and resume processing within a specified time. Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

### 6.6.4 Stress testing

Stress tests are designed to confront programs with abnormal situations. Stress testing executes a program in a manner that demands resources in abnormal quantity, frequency, or volume. For example, a test case that may cause

thrashing in a virtual operating system.

### 6.6.5 Performance Testing

For real time and embedded systems, performance testing is essential. In these systems, the compromise on performance is unacceptable. Performance testing is designed to test run-time performance of software within the context of an integrated system.

### 6.6.6 Acceptance testing

Acceptance testing involves planning and execution of functional tests, performance tests, and stress tests in order to demonstrate that the implemented system satisfies its requirements. Stress tests are performed to test the limitations of the systems. For example, a compiler may be tested to determine the effect of symbol table overflow.

Acceptance test will incorporate test cases developed during unit testing and integration testing. Additional test cases are added to achieve the desired level of functional, performance and stress testing of the entire system.

#### 6.6.7 Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers’ site. Alpha testing is often employed for off-the-shelf software as a form of internal

acceptance testing, before the software goes to beta testing.

#### 6.6.8 Beta testing

Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the company. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

### 6.7 White-box and black-box testing

White box and black box testing are terms used to describe the point of view a test engineer takes when designing test cases. Black box is an external view of the test object and white box, an internal view.

In recent years the term grey box testing has come into common usage. The typical grey box tester is permitted to set up or manipulate the testing environment, like seeding a database, and can view the state of the product after her actions, like performing a SQL query on the database to be certain of the values of columns. It can also be used of testers who know the internal workings or algorithm of the software under test and can write tests specifically for the anticipated results. For example, testing a data warehouse implementation involves loading the target database with information, and verifying the correctness of data population and loading of data into the correct tables.

## 6.7.1 White box testing

White box testing (also known as clear **box testing, glass box testing or structural testing**) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise all paths and determines the appropriate outputs. In electrical hardware testing every node in a circuit may be probed and measured, an example is in circuit test (ICT).

Since the tests are based on the actual implementation, if the implementation changes, the tests probably will need to also. For example ICT needs updates if component values change, and needs modified/new fixture if the circuit changes. This adds financial resistance to the change process, thus buggy products may stay buggy. Automated optical inspection (AOI) offers similar component level correctness checking without the cost of ICT fixtures, however changes still require test updates.

While white box testing is applicable at the unit, integration and system levels, it's typically applied to the unit. So while it normally tests paths within a unit, it can also test paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover an overwhelming number of test cases, it might not detect unimplemented parts of the specification or missing requirements. But you can be sure that all paths through the test object are executed.

Typical white box test design techniques include:

1. Control flow testing
2. Data flow testing

#### 6.7.2 Black Box testing

Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.

This method of test design is applicable to all levels of development - unit, integration, system and acceptance. The higher the level, and hence the bigger and more complex the box, the more we are forced to use black box testing to simplify. While this method can uncover unimplemented parts of the specification, you can't be sure that all existent paths are tested. Some common approaches of black box testing are equivalence class partitioning, boundary value analysis etc.

## 6.8Black box and white box testing compared

White box testing is concerned only with testing the software product; it cannot guarantee that the complete specification has been implemented. Black box testing is concerned only with testing the specification; it cannot guarantee that all parts of the implementation have been tested. Thus black box testing is testing against the specification and will discover faults of omission, indicating that part of the specification has not been fulfilled. White box testing is testing against the implementation and will discover faults of commission, indicating that part of the implementation is faulty. In order to fully test a software product both black and white box testing are required.

White box testing is much more expensive than black box testing. It requires the source code to be produced before the tests can be planned and is much more laborious in the determination of suitable input data and the determination if the software is or is not correct. The advice given is to start test planning with a black box test approach as soon as the specification is available. White box planning should commence as soon as all black box tests have been successfully passed, with the production of flow graphs and determination of paths. The paths should then be checked against the black box test plan and any additional required test runs determined and applied.

The consequences of test failure at this stage may be very expensive. A failure of a white box test may result in a change which requires all black box testing to be repeated and the re-determination of the white box paths. The cheaper option is to regard the process of testing as one of quality assurance rather than quality control. The intention is that sufficient quality will be put into all previous design and production stages so that it can be expected that testing will confirm that there are very few faults present, quality assurance, rather than testing being relied upon to discover any faults in the software, quality control. A combination of black box and white box test considerations is still not a completely adequate test rationale; additional considerations are to be introduced.

* 1. **Configuration management**

In Software Engineering Software Configuration Management is the task of tracking and controlling changes in the software part of the larger disciplinary field of Configuration Management.The goals of Software Configuration Management are generally Configuration, Identification, Configuration idioms and baselines, configuration control, implementing a control change process.The SCM practices include vision controls in the establishment of baselines. If something goes wrong, SCM can determine what was changed and who changed it.This is usually achieved by setting up a change control board whose primary function is to approve or reject all change request that is sent against any baseline. Configuration status accounting, reporting and recording all the necessary information on the status of the development process.

* 1. **SCM Features:**

1. **Enforcement:**With enforcement feature executing daily ensures that the system is configured to the desired state.
2. **Cooperating Enablement:** This feature helps to make the change configuration throughout the infrastructure with one change.
3. **Version Control Friendly:** With this feature, the user can take their choice of version for their work.
4. **Enable Change Control Processes:**As Software Configuration Management tools are version control and textual friendly we can make changes in code. Changes can be made as a merge request and send for review.
   1. **Small Questions**

**Q1. Do as directed.**

a) Write down the full form of COCOMO.

b) What is software engineering?

c) Define structured coding techniques.

d) What is the need of documentation?

e) Why software maintenance is required.

f) What are the features of good software?

g) What is software maintenance process?

h) Write down the objectives of testing.

i) Write about data structure oriented design.

j) Differentiate between white box and blackbox testing.

k) What do you mean by proto type?

l) What is meant by project size?

m) What is the need of creating a SRSdocument?

n) Define the term planning.

o) Define Halstead software science.

p) What do you mean by design errors?

q) Write down the disadvantages of usingprototyping approach.

r) Write down the advantages of usingCOCOMO for estimating cost?

* 1. **Short Answer Type Questions**

**Q2. Attempt any six questions.**

i) Briefly explain about requirement analysisand specifications.

ii) Explain about object oriented design.

iii) Explain about software coding and itsrequirements.

iv) Write short notes on:

a) Unit testing.

b) Integration testing.

v) What causes Bad SRS document.

vi) Differentiate between program andsoftware product.

vii) Explain about code walk through.

viii) Discuss about various issues related tosoftware coding and testing.

ix) Briefly discuss about system testing.

x) Differentiate between cohesion andcoupling.

xi) State the classification of coupling.

xii) Compare object oriented and functionoriented design.

xiii) What are the benefits of testing?

xiv) Explain in detail about 1S0 9000.

xv) Write short notes on

a) Lines of code b) Function point

* 1. **Long Type Questions**

Q.3 Explain briefly about the various software lifecycle models? Write about verification and

validation.

Q.4 What are the various techniques for projectestimation? Explain about each briefly.

Q.5 Explain about black box testing techniques.

Q.6 Write short note on:

a) Mcafe complexity.

b) White box testing.

Q.7 what is configuration management. Why it isrequired?