SPCA 210

# POSTGRADUATE COURSE

## M.C.A.- COMPUTER APPLICATION

SECOND YEAR

FOURTH SEMESTER

CORE PAPER - XIX

# SOFTWARE ENGINEERING

INSTITUTE OF DISTANCE EDUCATION

UNIVERSITY OF MADRAS

**M.C.A. COMPUTER APPLICATION** **CORE PAPER - XIX**

**IV - SEMESTER** **SOFTWARE ENGINEERING**

# WELCOME

Warm Greetings.

It is with a great pleasure to welcome you as a student of Institute of Distance Education, University of Madras. It is a proud moment for the Institute of Distance education as you are entering into a cafeteria system of learning process as envisaged by the University Grants Commission. Yes, we have framed and introduced Choice Based Credit System(CBCS) in Semester pattern from the academic year 2018-19. You are free to choose courses, as per the Regulations, to attain the target of total number of credits set for each course and also each degree programme. What is a credit? To earn one credit in a semester you have to spend 30 hours of learning process. Each course has a weightage in terms of credits. Credits are assigned by taking into account of its level of subject content. For instance, if one particular course or paper has 4 credits then you have to spend 120 hours of self-learning in a semester. You are advised to plan the strategy to devote hours of self-study in the learning process. You will be assessed periodically by means of tests, assignments and quizzes either in class room or laboratory or field work. In the case of PG (UG), Continuous Internal Assessment for 20(25) percentage and End Semester University Examination for 80 (75) percentage of the maximum score for a course / paper. The theory paper in the end semester examination will bring out your various skills: namely basic knowledge about subject, memory recall, application, analysis, comprehension and descriptive writing. We will always have in mind while training you in conducting experiments, analyzing the performance during laboratory work, and observing the outcomes to bring out the truth from the experiment, and we measure these skills in the end semester examination. You will be guided by well experienced faculty.

I invite you to join the CBCS in Semester System to gain rich knowledge leisurely at your will and wish. Choose the right courses at right times so as to erect your flag of success. We always encourage and enlighten to excel and empower. We are the cross bearers to make you a torch bearer to have a bright future.

With best wishes from mind and heart,

DIRECTOR

**M.C.A. COMPUTER APPLICATION** | **CORE PAPER - XIX**

**IV - SEMESTER** | **SOFTWARE ENGINEERING**

## COURSE WRITER

**Dr. K. Kalaiselvi**
Professor & Head of the Department
Vels Institute of Science Technology and Advanced Studies (VISTAS)
(Formerly Vels University)
Department of Computer Science
School of Computing Sciences
Pallavaram, Chennai - 600117
(Lesson 1-9)

**Dr. A. Akila**
Associate Professor
Department of Computer Science
School of Computing Sciences
Vels Institute of Science Technology and Advanced Studies (VISTAS)
Pallavaram, Chennai - 600117
(Lesson 10-15)

## EDITING & CO-ORDINATION

**Dr. S. Sasikala**
Associate Professor
Department of Computer Science
University of Madras
Institute of Distance Education
University of Madras
Chepauk, Chennai 600 005

# M.C.A. COMPUTER APPLICATION

## SECOND YEAR - FOURTH SEMESTER

## CORE PAPER - XIX

## SOFTWARE ENGINEERING

## SYLLABUS

**Unit 1:** Software Engineering - The nature of Software -Software Process Models- Waterfall Model-Incremental process models- Evolutionary process models-– Concurrent models-Specialized process models- Agile process –Agility principles

**Unit 2:** Requirements Engineering-Establishing the groundwork-Eliciting requirements-Building the Requirements Model-Validating Requirements – Requirements analysis-Modeling Approaches – Data Modeling Concepts- Modeling Strategies – Flow-Oriented Modeling-Behavioral Model.

**Unit 3:** Design concepts-The Design model-Architectural design-Component level design -User interface design-Software Configuration Management -The SCM Process- Version Control- Change Control- Configuration Audit

**Unit 4:** The Management spectrum – W5HH principle –Process and Project Metrics – Software Measurement – Software Project Estimation – Decomposition Techniques – Project Scheduling –Risk Management – Identification – Projection –Refinement- RMMM Plan.

**Unit 5:** Software Review Techniques:-Informal reviews-Formal Technical Reviews -Software Quality Assurance- SQA Tasks, Goals and Metrics- -Software Reliability - A Strategic Approach to Software Testing- Unit Testing- Integration Testing- Validation Testing - System Testing-The Art of Debugging – Software Maintenance

**Recommended Texts :**

1) Roger. S. Pressman, 2010, Software Engineering A Practitioner's approach, Seventh Edition, Tata McGraw-Hill, New Delhi.

**Reference Books:**

1) I. Sommerville, 2001, Software Engineering, 6th Edition, Addison Wesley, Boston.
2) Rajib Mal, 2005, -Fundamental of Software engineering , 2 nd Edition , PHI, New Delhi.
3) N. E. Fenton, S. L. Pfleenger, 2004, Software Metrics, Thomson Asia, Singapore.

# M.C.A. COMPUTER APPLICATION

**SECOND YEAR - FOURTH SEMESTER**

**CORE PAPER - XIX**

# SOFTWARE ENGINEERING

SCHEME OF LESSONS

# LESSON-1

# INTRODUCTION TO SOFTWARE ENGINEERING

## 1.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- ➢ Define the term ‘Software’.
- ➢ Understand the term ‘Software Engineering’ and its definitions given by the various authors.
- ➢ Understand about the principles of software and nature of software in detail.
- ➢ Understand about the different software used in different applications.
- ➢ Types of software myths and reality are explained.

### Structure

## 1.2 Introduction

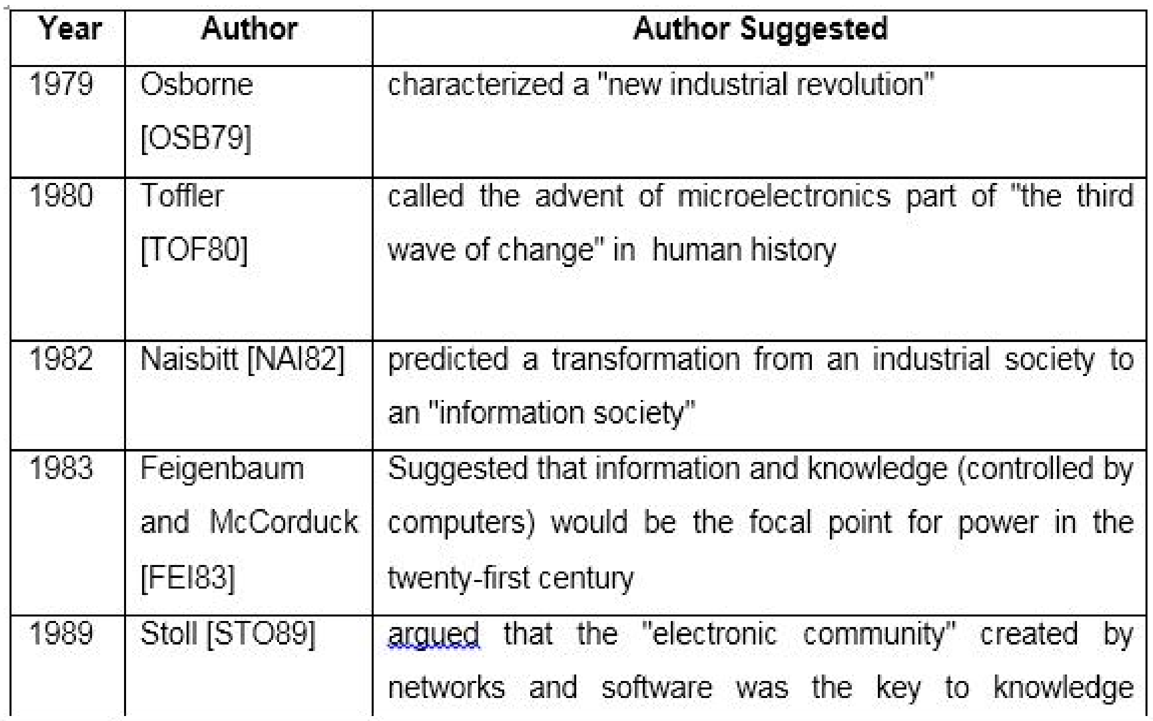
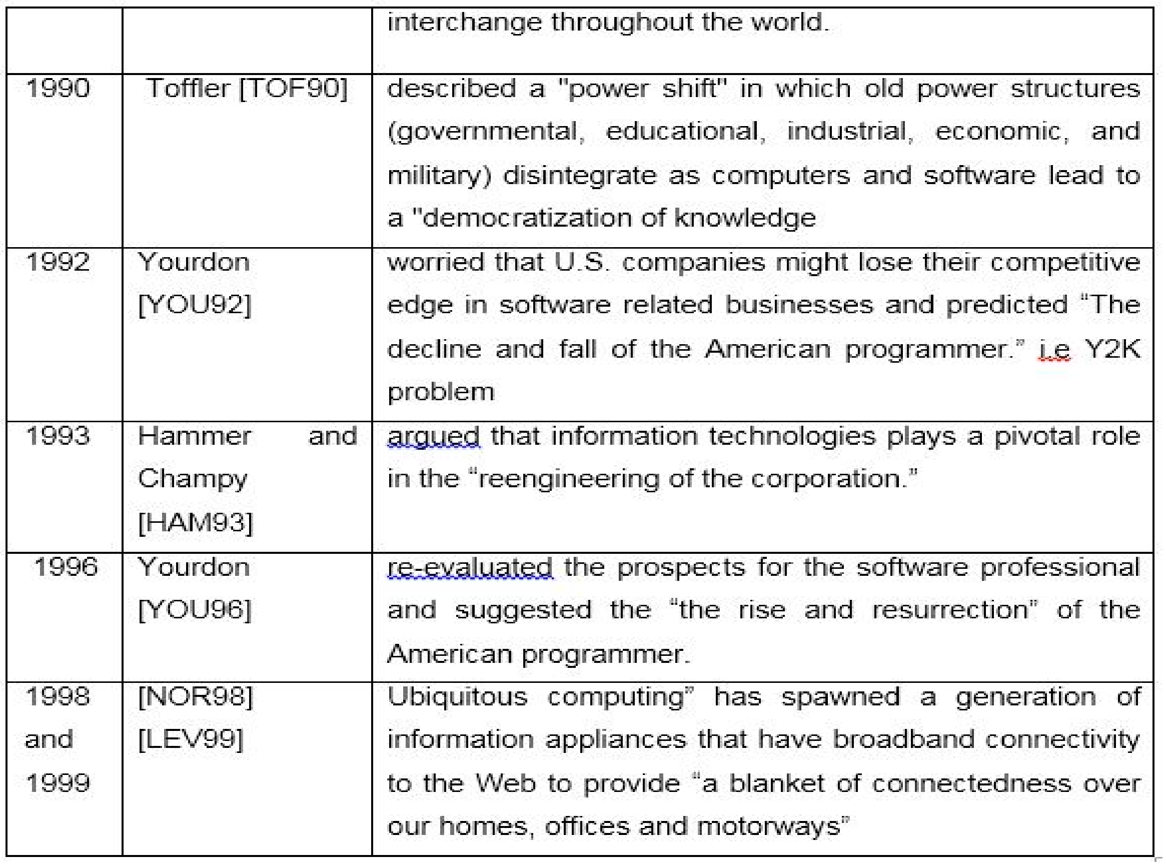
In this lesson we discuss about evolution of software during the period of 1970s to till date. It discusses about the term 'Software' and 'Software Engineering'. This lesson further discuss about the various software used in different applications like system software, real time software, embedded software, etc. It discuss about the software crisis and the software myths and its reality.

## 1.3 The Evolving Role of Software

Software plays a dual role. It acts as a product as well as vehicle for delivering a product. Product encompasses computing potential by computer hardware, network of computers that are accessible by local hardware. Software is an information transformer which can produce, manage, acquire, modify, display or transmit information as a single bit or complex as a multimedia presentation.

Further, Software acts as basis for controlling the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers product on time and transforms personal data which is useful in a local context, it manages business information, provides gateway to worldwide information networks (e.g. Internet) and provides the means for acquiring information in all different format.

**Table 1: Historical insight into changing perception of computers and software and their impact based on culture (period 1979-1999)**

| Year | Author | Author Suggested |
|---|---|---|
| 1979 | Osborne [OSB79] | characterized a "new industrial revolution" |
| 1980 | Toffler [TOF80] | called the advent of microelectronics part of "the third wave of change" in human history |
| 1982 | Naisbitt [NAI82] | predicted a transformation from an industrial society to an "information society" |
| 1983 | Feigenbaum and McCorduck [FEI83] | Suggested that information and knowledge (controlled by computers) would be the focal point for power in the twenty-first century |
| 1989 | Stoll [STO89] | argued that the "electronic community" created by networks and software was the key to knowledge interchange throughout the world. |
| 1990 | Toffler [TOF90] | described a "power shift" in which old power structures (governmental, educational, industrial, economic, and military) disintegrate as computers and software lead to a "democratization of knowledge |
| 1992 | Yourdon [YOU92] | worried that U.S. companies might lose their competitive edge in software related businesses and predicted "The decline and fall of the American programmer." i.e Y2K problem |
| 1993 | Hammer and Champy [HAM93] | argued that information technologies plays a pivotal role in the "reengineering of the corporation." |
| 1996 | Yourdon [YOU96] | re-evaluated the prospects for the software professional and suggested the "the rise and resurrection" of the American programmer. |
| 1998 and 1999 | [NOR98] [LEV99] | Ubiquitous computing" has spawned a generation of information appliances that have broadband connectivity to the Web to provide "a blanket of connectedness over our homes, offices and motorways" |

## 1.4 Software

Software encompasses the following:

- ➢ Instructions (computer programs) that when executed provide desired function and performance.
- ➢ Data structures that enable the programs to adequately manipulate information.
- ➢ Documents that describe the operation and use of the programs.
- ➢ Software = Program + Documentation + Operating Procedures

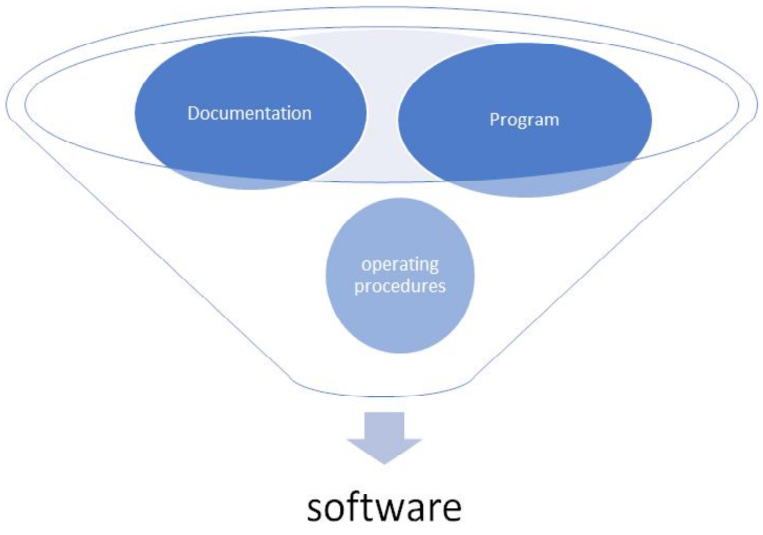


**Figure1.1 - Components of Software**

### 1.4.1 Software Characteristics

- Software is developed or engineered; it is not manufactured in the classical sense.
- Software doesn't "wear out" but it deteriorates (due to change). Hardware has bathtub curve of failure rate (high failure rate in the beginning, then drop to steady state, then cumulative effects of dust, vibration, abuse occurs).
- Although the industry is moving toward component-based construction (e.g. standard screws and off-the-shelf integrated circuits), most software continues to be custom-built. Modern reusable components encapsulate data and processing into software parts to be reused by different programs. E.g. graphical user interface, window, pull-down menus in library etc.

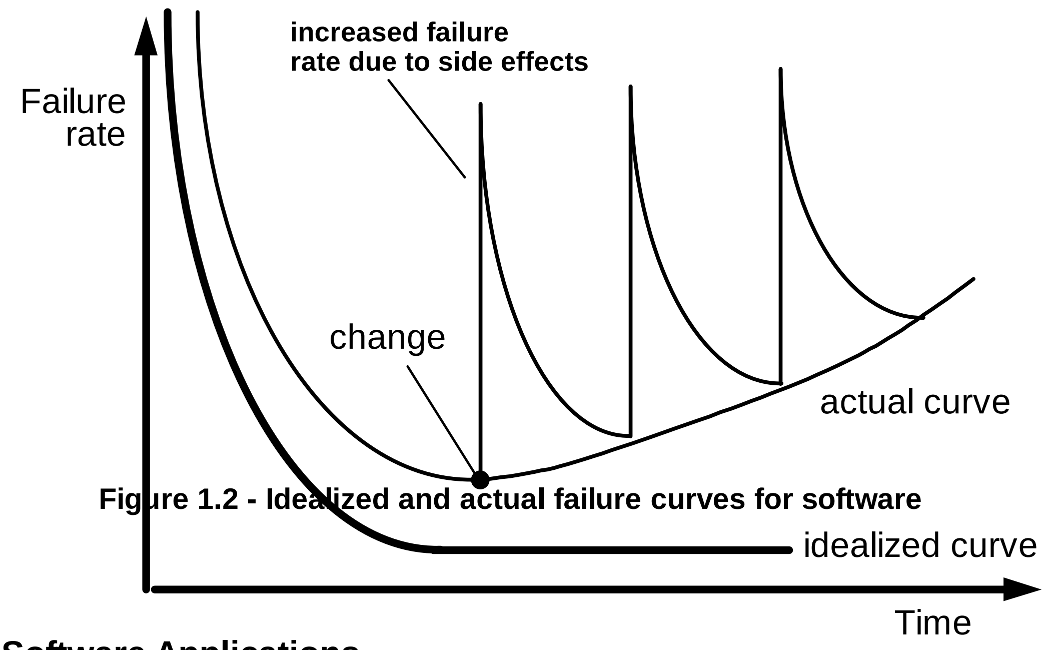


**Figure 1.2 - Idealized and actual failure curves for software**

### 1.4.2 Software Applications

- Software can be applied for a prespecified set of procedural steps or algorithm has been defined.
- Information content and determinacy are important factors in determining the nature of a software application.
- Content refers to the incoming and outgoing information.
- For example, many business applications use highly structured input data (a database) and produce formatted "reports." Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.
- Information determinacy refers to the predictability of the order and timing of information.
- An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm(s) without interruption, and produces resultant data in report or graphical format. Such applications are determinate.
- A multiuser operating system, on the other hand, accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions, and produces output that varies as a function of environment and time.

- Applications with these characteristics are indeterminate.
- It is somewhat difficult to develop meaningful generic categories for software applications.

Various types of Software applied in the different kind of applications are:

**1. System software**

- System software is a collection of programs written to service other programs.
- Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures.
- Other systems applications (e.g., operating system components, driver tele communications processors) process largely indeterminate data.

**2. Real-time software**

- Software that monitors/analyses/controls real-world events referred as real time.
- Real-time software include a

a. Data gathering component which collects and formats information from an external environment.

b. Analysis component which transforms information as required by the application.

c. Control/Output component which responds to the external environment.

d. Monitoring component that coordinates all other components which needs to be maintained.

**3. Business software**

- Information processing system in business is the largest software application area.
- Discrete “systems” (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS).
- Software applications restructure existing data and facilitate business operations or management decision making.

- In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point of sale transaction processing).

**4. Engineering and scientific software**

- Characterized by “number crunching” algorithms.
- Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.
- Modern applications within the engineering/scientific area are moving away from conventional numerical algorithms.
- Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

**5. Embedded software**

- Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets.
- Embedded software performs limited and esoteric functions (e.g., keypad control for a microwave oven) or provides significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

**6. Personal computer software**

- The personal computer software market has multiplied over the past two decades.
- Applications are word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access.

**7. Web-based software**

- WebApps (Web applications) network centric software.
- Since web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.

- The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).
- Network becomes a massive computer, provides unlimited software resource that can be accessed by anyone with a modem.

**8. Artificial intelligence software**

- Artificial intelligence (AI) software uses non-numerical algorithm to solve complex problem.
- Expert systems, also called knowledge based systems, Robotics, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

### 1.4.3 Software—New Categories

- Open world computing—pervasive, ubiquitous, distributed computing due to wireless networking.
- How to allow mobile devices, personal computer, enterprise system to communicate across vast network.
- Net sourcing—the Web as a computing engine. How to architect simple and sophisticated applications to target end-users worldwide.
- Open source— “free” source code open to the computing community.

Other software are

- Data mining
- Grid computing
- Cognitive machines
- Software for nanotechnologies

**What is software engineering?**

Software engineering is an engineering discipline which is concerned with all aspects of software production Software engineers should adopt a systematic and organised approach to their work; use appropriate tools and techniques depending on

• The problem to be solved.

• The development constraints and use the resources available.

**Definition**

**There are several definitions of Software Engineering**

- “Software Engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and with cost estimates”.
- **Stephen Schach** defined the same as “A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements”.
- **Boehm** defined as “Software engineering is the application of science and mathematics by which capabilities of computer equipment are made useful to man via computer programs, procedures and associated documents”.
- **Fritz Bauer** defined software engineering as “The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines”.
- The primary goals of software engineering are to improve the quality of software product and to increase the productivity and job satisfaction of software engineers.

## 1.5 Software: A crisis on the horizon?

- There seems to be an inordinate fascination with the spectacular software failures and a tendency to ignore the large number of successes achieved by the software industry.
- Software developers are capable of producing software that functions properly most of the time.

- ➢ The biggest problem facing modern software engineers is trying to figure out how to produce software fast enough to meet the growing demand for more products, while also having to maintain a growing volume of existing software.
- ➢ The word *crisis* is defined in *Webster's Dictionary* as "a turning point in the course of anything; decisive or crucial time, stage or event."
- ➢ Rather, the affliction encompasses problems associated with how we develop software, how we support a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

### Check your Progress

1. **Choose the best answer**:

Software is __________.

(a) Superset of programs (b) subset of programs

(c) Set of programs (d) none of the above

2. **Fill up the blanks**:

(a) Software consists of ____________________.

(b) The primary goals of software engineering are to improve the ______of software product.

3. **Say True or False**

(a) An efficient process is required to produce bad quality products.

(b) Artificial intelligence (AI) software uses non-numerical algorithm to solve complex problem.

## 1.6 Software MYTHS

- ✓ The word 'Myth' means a wrong belief in his information.
- ✓ Software myths are beliefs about software and the process used to build it.
- ✓ Many **software** problems arise due to **myths** that are formed during the initial stages of **software** development.

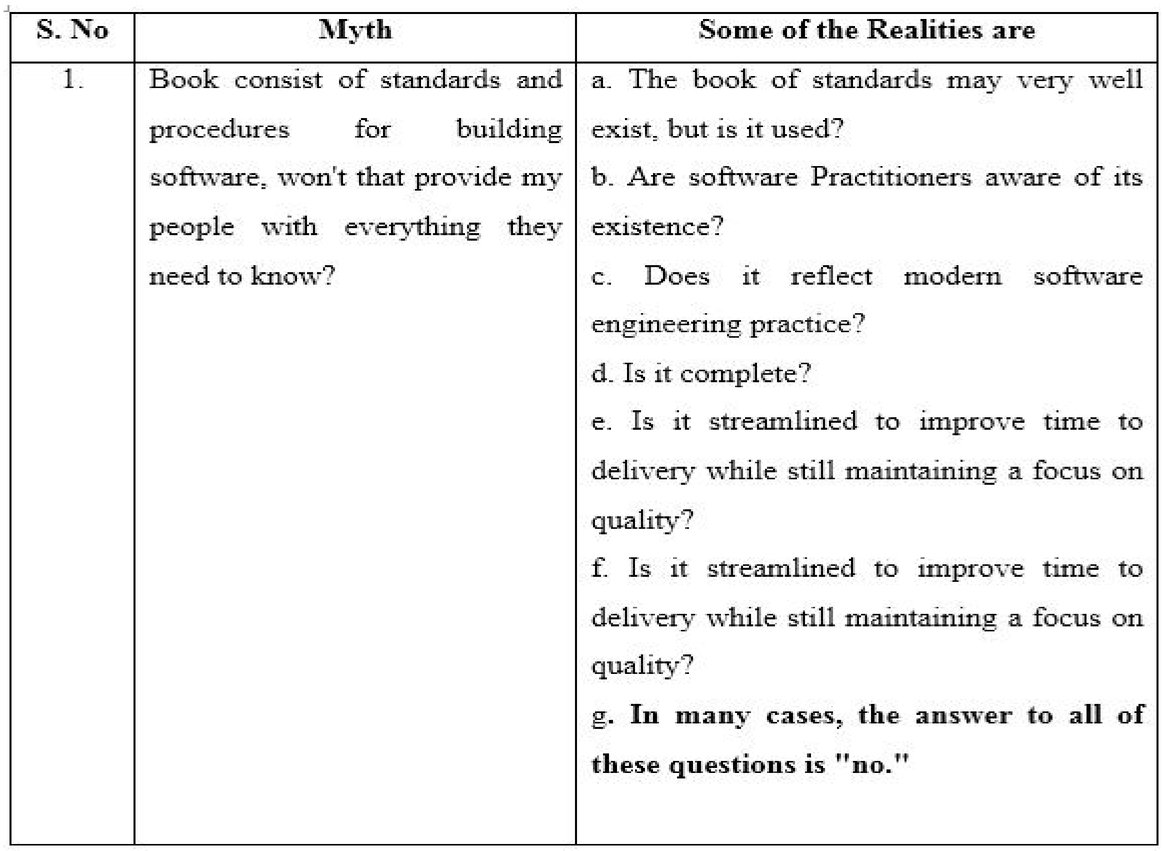
- ✓ Software myths propagated misinformation and confusion.
- ✓ Software myths had a number of attributes, appeared to be reasonable statements of fact (sometimes containing elements of truth), they had an intuitive feel, and they were often promulgated by experienced practitioners who “knew the score.”

**Types of Myths are:**

1. Management Myths
2. Customer Myths
3. Developer / Practitioner Myths

## 1.6.1 Management Myths

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).



| S. No | Myth | Some of the Realities are |
|---|---|---|
| 1. | Book consist of standards and procedures for building software, won't that provide my people with everything they need to know? | a. The book of standards may very well exist, but is it used?<br>b. Are software Practitioners aware of its existence?<br>c. Does it reflect modern software engineering practice?<br>d. Is it complete?<br>e. Is it streamlined to improve time to delivery while still maintaining a focus on quality?<br>f. Is it streamlined to improve time to delivery while still maintaining a focus on quality?<br>**g. In many cases, the answer to all of these questions is "no."** |

| | | |
|---|---|---|
| 2. | People have state-of-the-art software development tools, after all, we buy them the newest computers. | a. It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development.<br>b. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively. |
| 3. | If we get behind schedule, we can add more programmers and catch up<br>(Sometimes called the Mongolian horde concept). | a. Software development is not a mechanistic process like manufacturing.<br>b. In the words of Brooks [BRO75]: "adding people to a late software project makes it later."<br>c. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.<br>d. People can be added but only in a planned and well-coordinated manner. |
| 4. | If we decide to outsource the software project to a third party, we can just relax and let that firm build it. | If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects. |

### 1.6.2 Customer myths

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract.

In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

| S.No | Myth | Some of the Realities are |
|---|---|---|
| 1. | A general statement of objectives is sufficient to begin writing programs— we can fill in the details later. | a. A poor up-front definition is the major cause of failed software efforts.<br>b. A formal and detailed description of the information domain, function, behaviour, performance, interfaces, design constraints, and validation criteria is essential.<br>c. These characteristics can be determined only after thorough communication between customer and developer. |
| 2. | Project requirements continually change, but change can be easily accommodated<br>because software is flexible. | a. It is true that software requirements change, but the impact of change varies with the time at which it is introduced.<br>b. If serious attention is given to up-front definition, early requests for change can be accommodated easily. |

| | | |
|---|---|---|
| | | c. The customer can review requirements and recommend modifications with relatively little impact on cost.<br>d. When changes are requested during software design, the cost impact grows rapidly.<br>e. Resources have been committed and a design framework has been established.<br>f. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost.<br>g. Changes in function, performance, |
| | | interface, or other characteristics during implementation (code and test) have a severe impact on cost.<br>h. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier. |

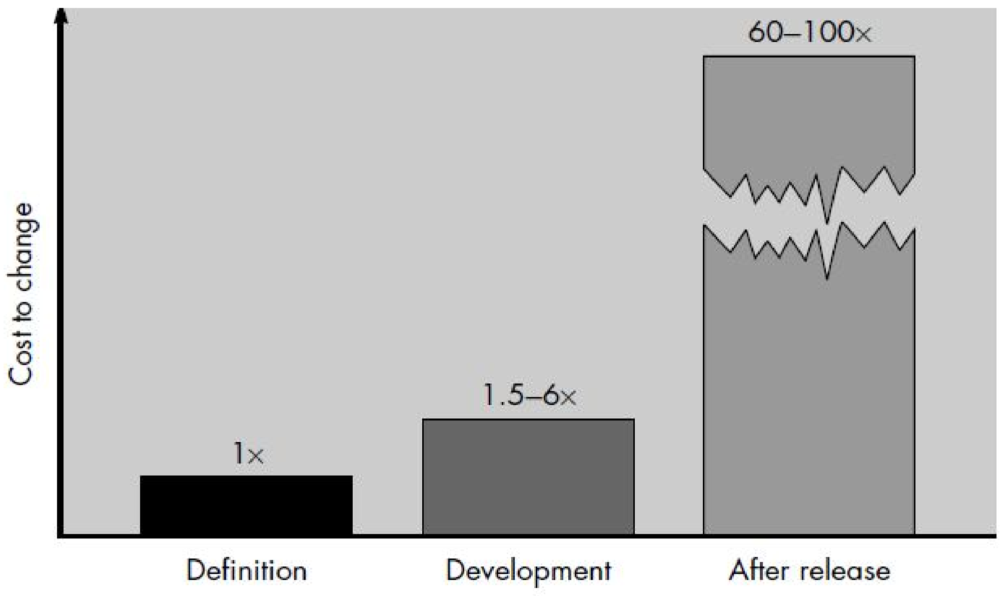


**Figure 1.3- The Impact of change**

### 1.6.3 Practitioner's myths

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software programming was viewed as an art form. Old ways and attitudes die hard.

| S.No | Myths | Some of the Realities are |
|---|---|---|
| 1. | Once we write the program and get it to work, our job is done. | a. Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." <br><br> b. Industry data indicate between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time. |
| 2. | Until we get the program "running" we have no way of assessing its quality. | a. One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review. |

| | | |
|---|---|---|
| | | b. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects. |
| 3. | The only deliverable work product for a successful project is the working program. | a. A working program is only one part of a software configuration that includes many elements.<br>b. Documentation provides a foundation for successful engineering and, more important, guidance for software support. |
| 4. | Software engineering will make us create voluminous and unnecessary documentation<br>and will invariably slow us down. | a. Software engineering is not about creating documents.<br>b. It is about creating Quality, better quality leads to reduced rework.<br>c. And reduced rework results in faster delivery times. |
| | | d. Many software professionals recognize the fallacy of the myths just described.<br>e. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach.<br>f. Recognition of software realities is the first step toward formulation of practical solutions for software engineering. |

## 1.7 Software Management Myths

Pressman describes managers beliefs in the following mythology as grasping at straws:

- ➢ Development problems can be solved by developing and documenting standards.
- ➢ Standards have been developed by companies and standards organizations are very useful.
- ➢ However, they are frequently ignored by developers because they are irrelevant and incomplete, and sometimes incomprehensible.
- ➢ Development problems can be solved by using state-of-the art tools.
- ➢ Tools may help, but there is no magic.
- ➢ Problem solving requires more than tools, it requires great understanding.
- ➢ As Fred Brooks (1987) says, there is no silver bullet to slay the software development werewolf.
- ➢ When schedules slip, just add more people. This solution seems intuitive: if there is too much work for the current team, just enlarge it.
- ➢ Unfortunately, increasing team size increases communication overhead.
- ➢ New workers must learn project details taking up the time of those who are already immersed in the project.
- ➢ Also, a larger team has many more communication links, which slows progress. Fred Brooks (1975) gives us one of the most famous software engineering maxims, which is not a myth, “adding people to a late project makes it later”.

### 1.7.1 Software Customer Myths

- ✓ Customers often vastly underestimate the difficulty of developing software.
- ✓ Sometimes marketing people encourage customers in their misbeliefs.
- ✓ Change is easily accommodated, since software is malleable.

- ✓ Software can certainly be changed, but often changes after release can require an enormous amount of labour.
- ✓ A general statement of need is sufficient to start coding.
- ✓ This myth reminds me of a cartoon that I used to post on my door.
- ✓ It showed the software manager talking to a group of programmers, with the quote: “You programmers just start coding while I go down and find out what they want the program to do”.
- ✓ However, for developers to have a chance to satisfy the customers’ requirements, they need detailed descriptions of these requirements.
- ✓ Developers cannot read the minds of customers.

### 1.7.2 Developer Myths

- ❖ Developers often want to be artists (or artisans), but the software development craft is becoming an engineering discipline.
- ❖ The job is done when the code is delivered.
- ❖ Commercially successful software may be used for decades.
- ❖ Developers must continually maintain such software: they add features and repair bugs.
- ❖ Maintenance costs predominate over all other costs; maintenance may be 70% of the development costs.
- ❖ This myth is true only for shelf ware — software that is never used, and there are no customers for next release of a shelf ware product.
- ❖ Project success depends solely on the quality of the delivered program.
- ❖ Documentation and software configuration information is very important to the quality.
- ❖ After functionality, maintainability, sees the preceding myth, is of critical importance. Developers must maintain the software and they need good design documents, test data, etc to do their job.
- ❖ You can’t assess software quality until the program is running.

- ❖ There are static ways to evaluate quality without running a program.
- ❖ Software reviews can effectively determine the quality of requirements documents, design documents, test plans, and code. Formal (mathematical) analyses are often used to verify safety critical software, software security factors, and very-high reliability software.

## 1.7 Summary

Software has become the key element in the evolution of computer-based systems and products. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. But early "programming" culture and history have created a set of problems that persist today. Software has become the limiting factor in the continuing evolution of computer based systems. Software is composed of programs, data, and documents. Each of these items comprises configuration that is created as part of the software engineering process. The intent of software engineering is to provide a framework for building software with higher quality.

## 1.8 Check your Answers

1. (c) Set of programs

2. a. Program + Documentation + Operating Procedures

2. b Quality

3. a. False

3. b.True

## 1.9 Model Questions

1. Define Software.

2. Write briefly about the software crisis .

3. What are the various software available?

4. What is software engineering?

5. Write briefly about the software myths.

# LESSON -2

# SOFTWARE PROCESS

## 2.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- Understand what is Software Process?
- Understand the different types of Software Process models.
- Understand about the Fourth Generation Techniques .
- Understand the terminologies ‘Process’ and ‘Product’ in Software Engineering.

### Structure

## 2.2 Introduction

In this lesson, it discuss about the term ‘software process’ and further discuss about the types of software process models. It explains about the various types of software process models and its advantages and disadvantages are discussed. It helps to learn the difference between the term ‘process’ and ‘product’.

## 2.3 Software Process

A software process can be characterized as shown in Figure 2.1. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets, each a collection of software engineering work tasks, project milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model.

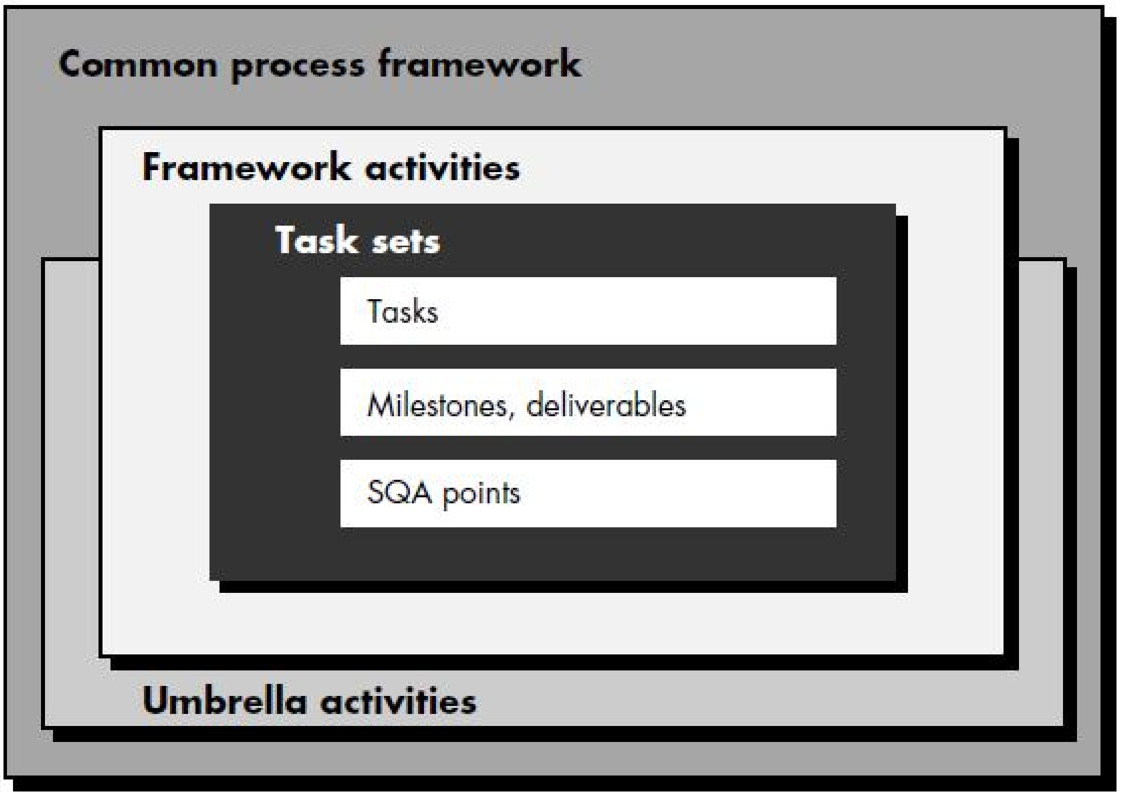


**Figure 2.1 - Common Process Framework**

Umbrella activities are independent of any one framework activity and occur throughout the process. In recent years, there has been a significant emphasis on “process maturity”. The **Software Engineering Institute (SEI)** has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organization search different levels of process maturity. To determine an organization current state of process maturity, the SEI uses an assessment that results in a five point grading scheme. The grading scheme determines compliance with a capability maturity model (CMM) [PAU93] that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company’s software engineering practices and establishes five process maturity levels that are defined in the following manner:

**Level 1: Initial.** The software process is characterized as ad-hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

**Level 2: Repeatable.** Basic project management processes are established to track cost, schedule, and functionality. The necessary process is in place to repeat earlier successes on projects with similar applications.

**Level 3: Defined**. The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

**Level 4: Managed.** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

**Level 5: Optimizing.** Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4. The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

The SEI has associated key process areas (KPAs) with each of the maturity levels. The KPAs describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics:

• **Goals**—the overall objectives that the KPA must achieve.

• **Commitments**—requirements (imposed on the organization) that must be met to achieve the goals or provide proof of intent to comply with the goals.

• **Abilities**—those things that must be in place (organizationally and technically) to enable the organization to meet the commitments.

• **Activities**—the specific tasks required to achieve the KPA function.

• **Methods for monitoring implementation**—the manner in which the activities are monitored as they are put into place.

• **Methods for verifying implementation**—the manner in which proper practice for the KPA can be verified.

Eighteen KPAs (each described using these characteristics) are defined across the maturity model and mapped into different levels of process maturity. The following KPAs should be achieved at each process maturity level: 3

**Process maturity level 2**

- Software configuration management
- Software quality assurance
- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

**Process maturity level 3**

- Peer reviews
- Intergroup coordination
- Software product engineering
- Integrated software management
- Training program
- Organization process definition
- Organization process focus

**Process maturity level 4**

- Software quality management
- Quantitative process management

**Process maturity level 5**

- Process change management
- Technology change management
- Defect prevention

Each of the KPAs is defined by a set of key practices that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted. The SEI defines key indicators as “those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved.” Assessment questions are designed to probe for the existence (or lack thereof) of a key indicator.

## 2.4 Software Process Models

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers are discussed. This strategy is often referred to as a process model or a software engineering paradigm. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

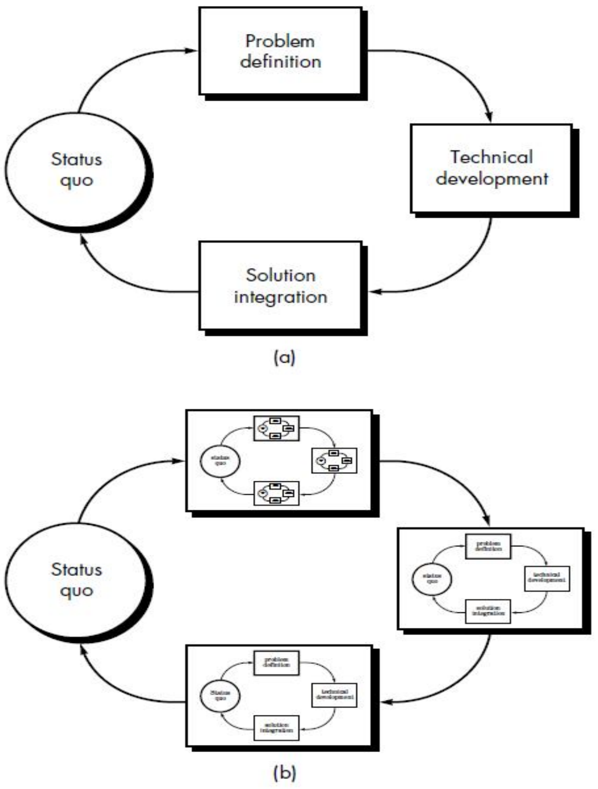


**Figure 2.2 (a) - The Phases of a problem solving loop**

**Figure 2.2 (b) - The phases within phases of the problem solving loop**

## 2.5 The Linear Sequential Model

The linear sequential model is referred as classic life cycle or the waterfall model. Many people dismiss the waterfall as obsolete and it certainly does have problems. But this model can still be used in some situations. Among the problems that are encountered sometimes when the *waterfall* model is applied are:

- A Real project rarely follows the sequential flow that the model proposes. Change can cause confusion as the project proceeds.
- It is difficult for the customer to state all the requirements explicitly. The waterfall model requires such demand.
- The customer must have patience. A working of the program will not be available until late in the project time-span.

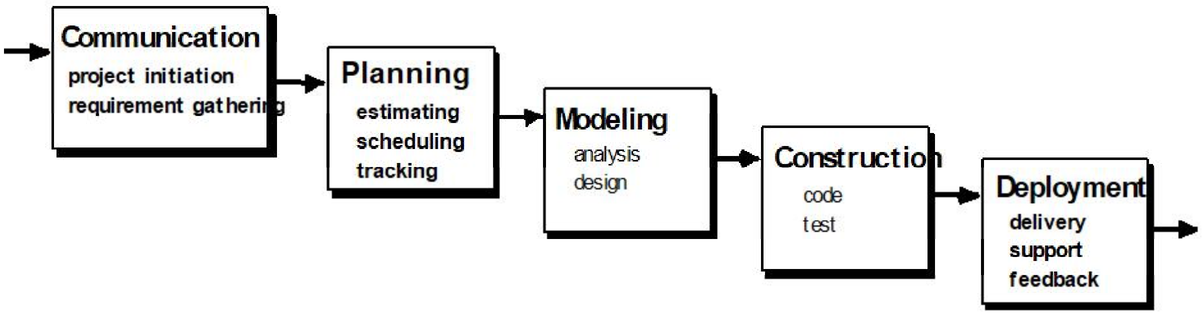


**Figure 2.1. Linear Sequential Model/ Waterfall Model**

**Waterfall Advantages**

- Natural approach for problem solving
- Conceptually simple, cleanly divides the problem into distinct independent phases
- Easy to administer in a contractual setup – each phase is a milestone

**Waterfall disadvantages**

- Assumes that requirements can be specified and frozen early.
- May fix hardware and other technologies too early.
- Follows the “big bang” approach – all or nothing delivery; too risky.
- Very document oriented, requiring docs at the end of each phase.

## 2.6 The Proto Typing Model

Prototyping is defined as the process of developing a working replication of a product or system that has to be engineered. The Prototyping Model is one of the most popularly used Software Development Life Cycle Models (SDLC models). This model is used when the customers do not know the exact project requirements beforehand.

In this model, a prototype of the end product is first developed, tested and refined as per customer feedback repeatedly till a final acceptable prototype is achieved which forms the basis for developing the final product.

In this process model, the system is partially implemented before or during the analysis phase thereby giving the customers an opportunity to see the product early in the life cycle. The process starts by interviewing the customers and developing the incomplete high-level paper model. This document is used to build the initial prototype supporting only the basic functionality as desired by the customer. Once the customer figures out the problems, the prototype is further refined to eliminate them. The process continues till the user approves the prototype and finds the working model to be satisfactory.

There are 2 approaches for this model:

**1. Rapid Throwaway Prototyping**

This technique offers a useful method of exploring ideas and getting customer feedback for each of them. In this method, a developed prototype need not necessarily be a part of the ultimately accepted prototype. Customer feedback helps in preventing unnecessary design faults and hence, the final prototype developed is of a better quality.

**2. Evolutionary Prototyping**

In this method, the prototype developed initially is incrementally refined on the basis of customer feedback till it finally gets accepted. In comparison to Rapid Throwaway Prototyping, it offers a better approach which saves time as well as effort. This is because developing a prototype from scratch for every iteration of the process can sometimes be very frustrating for the developers.

## 2.7 The Rad Model

RAD model is Rapid Application Development model. It is a type of incremental model. In RAD model the components or functions are developed in parallel as if they were mini projects. The developments are time boxed, delivered and then assembled into a working prototype. This can quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements.

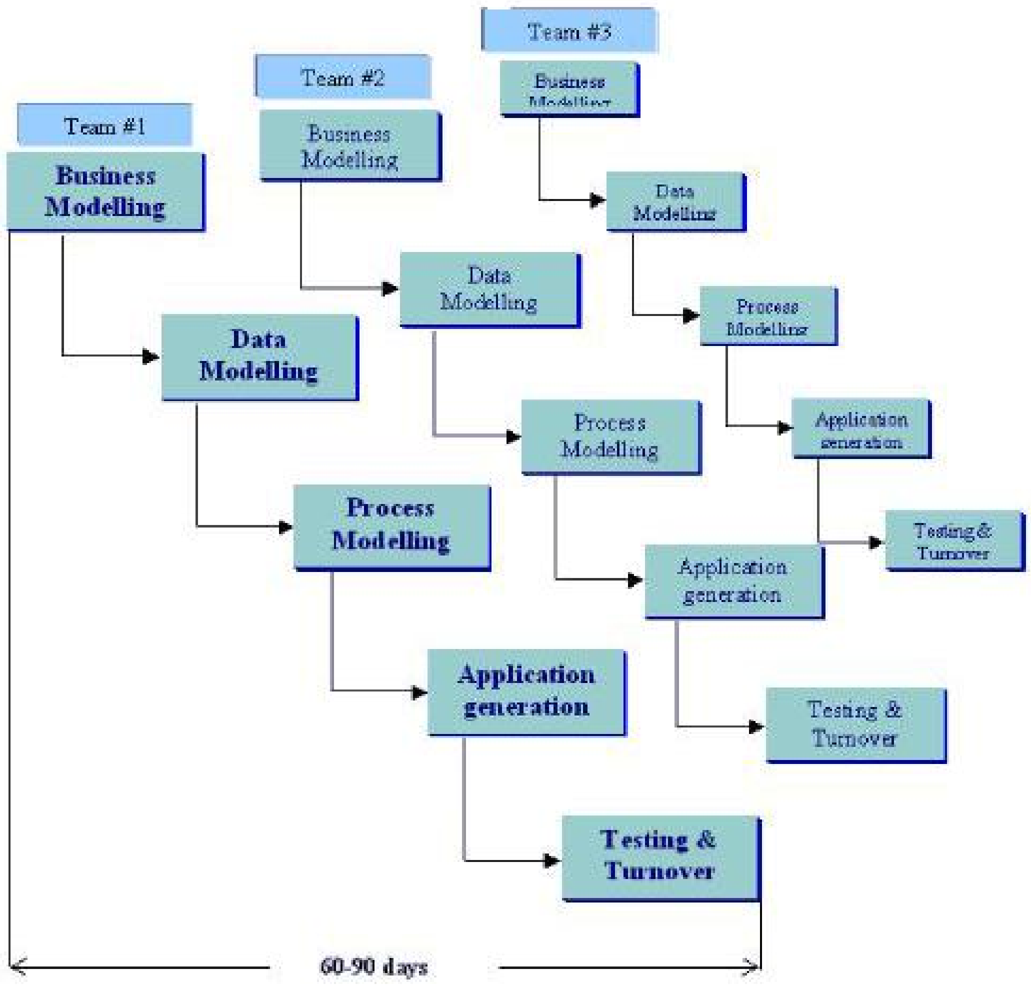


**Figure 2.3 . RAD-Model**

The phases in the rapid application development (RAD) model are:

**Business modeling:**

The information flow is identified between various business functions.

**Data modeling:**

Information gathered from business modeling is used to define data objects that are needed for the business.

**Process modeling:**

Data objects defined in data modeling are converted to achieve the business information flow to achieve some specific business objective. Description is identified and created for CRUD of data objects.

**Application generation:**

Automated tools are used to convert process models into code and the actual system.

**Testing and turnover:**

Test new components and all the interfaces.

## 2.8 Evolutionary Software Process Models

### 2.8.1 The Incremental Model

Incremental Model is a process of software development where requirements are broken down into multiple standalone modules of software development cycle. Incremental development is done in steps from analysis design, implementation, testing/verification, maintenance.

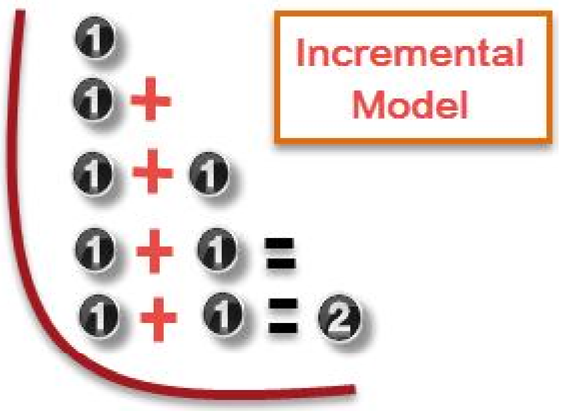


**Figure 2. 4 Steps to Incremental Model**

Each iteration passes through the **requirements, design, coding and testing phases**. And each subsequent release of the system adds function to the previous release until all designed functionality has been implemented.

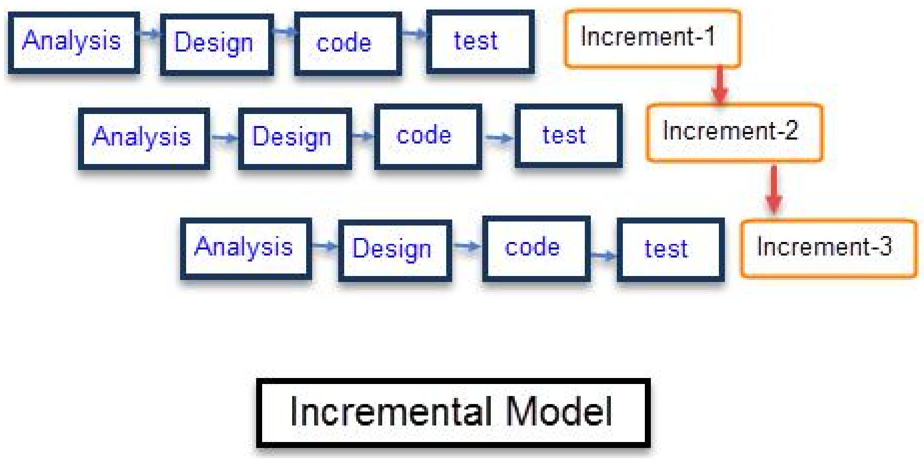


**Figure 2.5 Description of Incremental Model**

The system is put into production when the first increment is delivered. The first increment is often a core product where the basic requirements are addressed, and supplementary features are added in the next increments. Once the core product is analyzed by the client, there is plan development for the next increment.

**Characteristics of an Incremental module includes**

- System development is broken down into many mini development projects.
- Partial systems are successively built to produce a final total system.
- Highest priority requirement is tackled first.
- Once the requirement is developed, requirement for that increment are frozen.

### 2.8.2The Spiral Model

**Spiral model** is one of the most important Software Development Life Cycle models, which provides support for **Risk Handling**. In its diagrammatic representation, it looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. **Each loop of the spiral is called a Phase of the software development process.** The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks. As the project manager dynamically determines the number of phases, so the project manager has an important role to develop a product using spiral model.

The Radius of the spiral at any point represents the expenses (cost) of the project so far, and the angular dimension represents the progress made so far in the current phase.

Each phase of Spiral Model is divided into four quadrants as shown in the above figure. The functions of these four quadrants are discussed below-

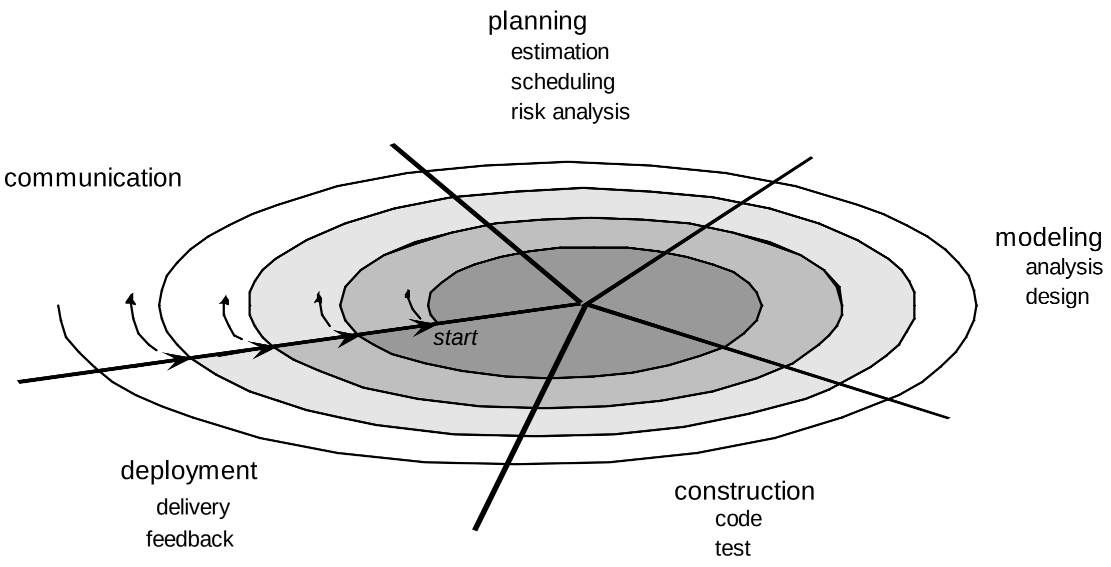


**Figure 2.4 . Spiral Model**

**1. Objectives determination and identify alternative solutions:** Requirements are gathered from the customers and the objectives are identified, elaborated and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.

**2. Identify and resolve Risks:** During the second quadrant all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution is identified and the risks are resolved using the best possible strategy. At the end of this quadrant, Prototype is built for the best possible solution.

**3. Develop next version of the Product:** During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available.

**4. Review and plan for the next Phase:** In the fourth quadrant, the Customers evaluate the so far developed version of the software. In the end, planning for the next phase is started.

## 2.8.3 The WINWIN Spiral Model

The WinWin spiral software engineering methodology [#!WinWin!#] is a recent example of a software engineering methodology. The WinWin spiral software engineering methodology expands the Boehm-Spiral methodology by adding a priority setting step, the WinWin process, at the beginning of each spiral cycle and by introducing intermediate goals, called anchor points.

The WinWin process identifies a decision point. For each decision point, the objectives, constraints, and alternatives are established and a WinWin condition is established. This may require a negotiation among the stakeholders and some reconciliation. The anchor points establish three intermediate goals. The first anchor point, called the life cycle objective (LCO), establishes sound business cases for the entire system by showing that there is at least one feasible architecture that satisfies the goals of the system.

The first intermediate goal is established when the top-level system objectives and scope, the operational concepts, the top-level system requirements, architecture, life cycle model, and system prototype are completed. This first anchor point establishes the why, what, when, who, where, how, and cost of the system. At the completion of this anchor point, a high level analysis of the system is available.

The second anchor point, called the life cycle architecture (LCA), defines the life cycle architecture. The third anchor point, called the initial operational capability (IOC), defines the operational capability, including the software environment needed for the first product release, operational hardware and site environment, and customer manuals and training. These two anchor points expand the high level analysis into other life cycle stages.

The WinWin spiral software engineering methodology is similar to the Water Sluice. The WinWin process could be considered a Water Sluice priority function, while the anchor points could represent Water Sluice stages. The WinWin process does not explicitly include non-monotonic effects. The anchor points are like the major stages in the life cycle of a product: initial development, deployment, operations, maintenance, legacy, and final discontinuation. The first anchor point is close to initial development. The second anchor point initiates deployment, while the third anchor point starts operations and maintenance.

### 2.8.4 The Concurrent Development Model

The concurrent process model defines a series of events that will trigger transition from state to state for each of the software engineering activities. For e.g., during early stages of design an inconsistency in the analysis model is uncovered.

This generates the event analysis model correction, which will trigger the analysis activity from the **done** state into the **awaiting Changes** State.

The concurrent process model is often used as the paradigm for the development of client/server applications. A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions a system dimension and component dimension. System level issues are addressed using three activities, design assembly, and use. The component dimension addressed with two-activity design and realization.

Concurrency is achieved in two ways:

(1) System and component activities occur simultaneously and can be modeled using the state – oriented approach.

(2) A typical client server application is implemented with many components, each of which can be designed and realized concurrently.

The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software-engineering activities to a sequence of events, it defines a network of activities. Each activity on the network exists simultaneously with other activities. Events generated within a given activity or at some other place in the activity network trigger transitions among the states of an activity

## 2.9 Component-based Development

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure **component reusability**. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit. There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services.

These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.

Component-oriented software design has many advantages over the traditional object-oriented approaches such as:

- Reduced time in market and the development cost by reusing existing components.
- Increased reliability with the reuse of the existing components.

### What is a Component?

A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.

A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

### Views of a Component

A component can have three different views “ object-oriented view, conventional view and process-related view.

### Object-oriented view

A component is viewed as a set of one or more cooperating classes. Each problem domain class (analysis) and infrastructure class (design) are explained to identify all attributes and operations that apply to its implementation. It also involves defining the interfaces that enable classes to communicate and cooperate.

**Conventional view**

It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.

**Process-related view**

In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.

- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

## 2.10 The Formal Methods Model

The **Formal Methods Model** is concerned with the application of a mathematical technique to design and implement the software. This model lays the foundation for developing complex system and supporting the program development. The formal methods used during the development process provide a mechanism for eliminating problems, which are difficult to overcome using other software process models. The software engineer creates formal specifications for this model. These methods minimize specification errors and this result in fewer errors when the user begins using the system.

Formal methods comprise formal specification using mathematics to specify the desired properties of the system. Formal specification is expressed in a language whose syntax and semantics are formally defined. This language comprises a syntax that defines specific notation used for specification representation; semantic, which uses objects to describe the system; and a set of relations, which uses rules to indicate the objects for satisfying the specification.

Generally, the formal method comprises two approaches, namely, property based and model-based. The **property-based specification** describes the operations performed on the system. In addition, it describes the relationship that exists among these operations. A property-based specification consists of two parts: signatures, which determine the syntax of operations and an equation, which defines the semantics of the operations through a set of equations known as **axioms.**

The **model-based specification** utilizes the tools of set theory, function theory, and logic to develop an abstract model of the system. In addition, it specifies the operations performed on the abstract model. The model thus developed is of a high level and idealized. A model-based specification comprises a definition of the set of states of the system and definitions of the legal operations performed on the system to indicate how these legal operations change the current state.

Various advantages and disadvantages associated with a formal method model are listed in Table.

**Table 3.1 Advantages and Disadvantages of Formal Methods Model**

| Advantages | Disadvantages |
|---|---|
| 1. Discovers ambiguity, incompleteness, and inconsistency in the software.<br>2. Offers defect-free software.<br>3. Each iteration incrementally grows with an effective solution.<br>4. This model does not involve high complexity rate.<br>5. Formal specification language semantics verify self-consistency. | 1. Time consuming and expensive.<br>2. Difficult to use this model as a communication mechanism for non-technical personnel.<br>3. Extensive training is required since only few developers have the essential knowledge to implement this model. |

## 2.11 Fourth Generation Techniques

Implementation using a **4GL** (4th Generation Techniques) enables the software developer to represent desired results in a manner that leads to automatic generation of code to create those results. Obviously, a data structure with relevant information must exist and be readily accessible by the 4GL. To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

**Software development environment that supports the 4GT paradigm includes some or all of the following tools:**

1) Non-procedural languages for database query

2) Report generation

3) Data manipulation

4) Screen interaction and definition

5) Code generation and High-level graphics capability

6) Spreadsheet capability

7) Automated generation of HTML and similar languages used for Web-site creation using advanced software tools.

**Advantages:**

**Simplified the programming process**

- Use non-procedural languages that encourage users and programmers to specify the results they want, while the computers determines the sequence of instruction that will accomplish those results.
- Use natural languages that impose no rigid grammatical rules.

**Disadvantages:**

- Less flexible that other languages.
- Programs written in 4GLs are generally far less efficient during program.
- Programs executed in high-level languages.

### Check your Progress

1. ___________ consists of natural approach for problem solving.

2. **Match the following**

| | |
|---|---|
| a) Formal Methods Model | (i) decomposition of the design |
| b) Component based architecture | (ii) identifies a decision point |
| c) WinWin process | (iii) mathematical technique |
| d) Rapid Application Development | (iv) problem divided into distinct independent phases |
| e) Waterfall model | (v) developments are time boxed |

3. **4GT paradigm includes** _________ for database query.

4. The primary objective of component-based architecture is to ensure _______.

5. **Say True or False**

   Each loop of the spiral is called a Phase of the software development process.

## 2.12 Process and Product

### Process

A software process (also known as software methodology) is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or, modifying an existing system.

Any software process must include the following four activities:

1. **Software specification** (or requirements engineering): Define the main functionalities of the software and the constrains around them.

2. S**oftware design and implementation**: The software is to be designed and programmed.

3. **Software verification and validation**: The software must conforms to it's specification and meets the customer needs.

4. **Software evolution** (software maintenance): The software is being modified to meet customer and market requirements changes.

**Product**

The product is the result of a project. The desired product satisfies the customers and keeps them coming back for more. Sometimes, however, the actual product is something less. The product pays the bills and ultimately allows people to work together in a process and build software.

Always keep the product in focus. Our current emphasis on process sometimes causes us to forget the product. This results in a poor product, no money, no more business, and no more need for people and process.

Instead of discussing different types of products (compilers, word processors, operating systems), I want to focus on two other aspects of the product. These are (1) the difficulty of the product and (2) the external and internal quality.

Difficult products demand process models that allow for experimenting and learning. Easy products call for process models that are simple, straightforward, and efficient (like the waterfall). Difficult products become easier when you bring in people with knowledge of the product.

Keep an eye on the external and internal quality of the product. External quality is what the customer sees. The customer is happy if the product has all the required functions, is easy to learn and use, runs quickly, and doesn't require much disk space and memory.

Internal quality is what the builder sees. High internal quality indicates, among other things, that the design and code are easy to understand and the software is modifiable and portable. When your customer announces he or she is changing hardware platforms and operating environments, high internal quality lets you change the product quickly and easily. These quality factors also influence the people and process. If portability (internal quality) is important, you need Unix and Macintosh people as well as MS-DOS/MS- Windows people on the project. If these people are not available, you must allow for learning and risk in your process.

## 2.13 Summary

In Software engineering, it integrates process, methods, and tools for the development of computer software. A number of different process models for software engineering have been proposed, each exhibiting strengths and weaknesses, but all having a series of generic phases in common. The principles, concepts and methods that enable us to perform the process.

- **Process** – method for doing something
  - Process typically has stages, each stage focusing on an identifiable task
  - Stages have methodologies
- **Software process** - methods for developing software
  - Best to view it as comprising of multiple processes

## 2.14 Check your Answers

**1.**water fall model

2. **Match the following**

| | |
|---|---|
| a) Formal Methods Model | (i) mathematical technique |
| b) component based architecture | (ii) decomposition of the design |
| c) WinWin process | (iii) identifies a decision point |
| d) Rapid Application Development | (iv) developments are time boxed |
| e) Waterfall model | (v) problem divided into distinct independent phases |

3. Non-procedural languages

4. component reusability

5. True

## 2.15 Model Questions

1. What is software process Model?
2. Write briefly about the prototype model.
3. What is the difference between process and product?
4. What are the advantages and disadvantages in fourth generation techniques?
5. Explain about the Evolutionary Software Process Models with neat diagram.

# LESSON-3

# AGILE DEVELOPMENT

## 3.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- ➢ Define the term ‘Agility’.
- ➢ Understand the Agile process and its agile software development strategy.
- ➢ Understand about the two principles of Agile viz., Agile Principle I and Agile Principle II in detail.
- ➢ Understand about the different software used in agile methodology namely, Extreme Programming (XP), Adaptive Software Development (ASD), Dynamic Systems Development Method

### Structure

## 3.2 Introduction

In this lesson, it discuss about the term ‘agility’ which is effective response to change. It discuss about the agile software development strategy. It further discusses about the Agility Principles I and II. It explains about the Extreme Programming in detail.

## 3.3 The Manifesto for Agile Software Development

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

**What is “Agility”?**

- Effective (rapid and adaptive) response to change (team members, new technology, requirements)
- Effective communication in structure and attitudes among all team members, technological and business people, software engineers and managers.
- Drawing the customer into the team. Eliminate “us and them” attitude.
- Planning in an uncertain world has its limits and plan must be flexible.

- Organizing a team so that it is in control of the work performed
- Eliminate all but the most essential work products and keep them lean.
- Emphasize an incremental delivery strategy as opposed to intermediate products that gets working software to the customer as rapidly as feasible.

***Rapid, incremental delivery of software***

- The development guidelines, stress delivery over analysis and design although these activates are not discouraged, and active and continuous communication between developers and customers.

**Why?**

The modern business environment is fast-paced and ever-changing. It represents a reasonable alternative to conventional software engineering for certain classes of software projects. It has been demonstrated to deliver successful systems quickly.

**What?**

Agile can be termed as “software engineering lite”. The basic activities- communication, planning, modeling, construction and deployment remain. But they morph into a minimal task set that push the team toward construction and delivery sooner only really important work product is an operational “software increment” that is delivered.

**Agility and the Cost of Change**

- Conventional wisdom is that the cost of change increases nonlinearly as a project progresses. It is relatively easy to accommodate a change when a team is gathering requirements early in a project. If there are any changes, the costs of doing this work are minimal. But if in the middle of validation testing, a stakeholder is requesting a major functional change. Then the change requires a modification to the architectural design, construction of new components, changes to other existing components, new testing and so on. Costs escalate quickly.
- A well-designed agile process may “flatten” the cost of change curve by coupling incremental delivery with agile practices such as continuous unit testing and pair programming. Thus team can accommodate changes late in the software project without dramatic cost and time impact.

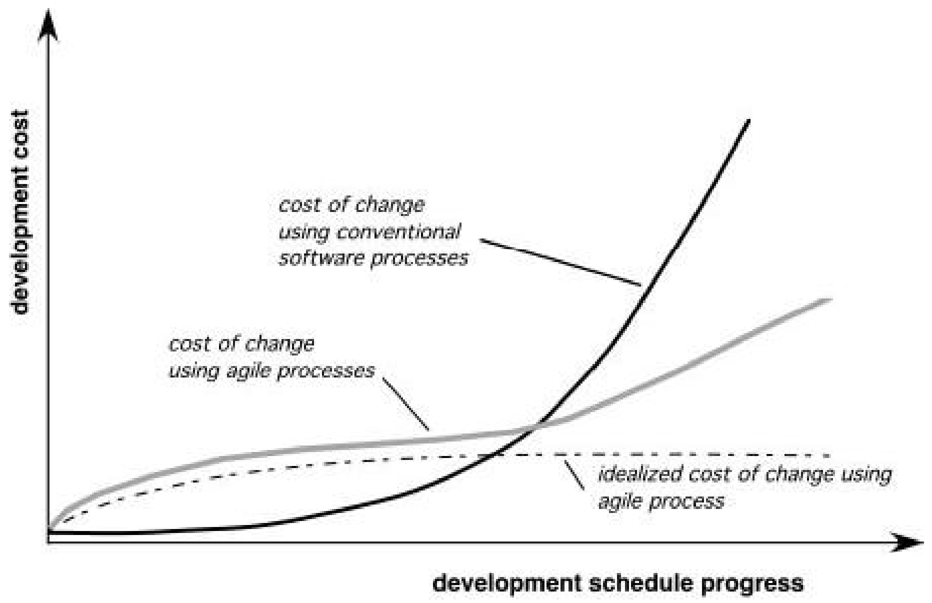


**Figure 3.1 Agility And The Cost Of Change**

## 3.4 An Agile Process

**Is driven by customer descriptions of what is required**

- Recognizes that plans are short-lived (some requirements will persist, some will change. Customer priorities will change)
- Develops software iteratively with a heavy emphasis on construction activities (design and construction are interleaved, hard to say how much design is necessary before construction. Design models are proven as they are created. )
- Analysis, design, construction and testing are not predictable.

***Thus has to Adapt as changes occur due to unpredictability***

- Delivers multiple 'software increments', deliver an operational prototype or portion of an OS to collect customer Feedback for adaption.

### 3.5 Agility Principles - I

1. The highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Changing requirements are welcomed, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face–to–face conversation.

## 3.6 Agility Principles - II

1. Working software is the primary measure of progress.
2. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
3. Continuous attention to technical excellence and good design enhances agility.
4. Simplicity – the art of maximizing the amount of work not done is essential.
5. The best architectures, requirements, and designs emerge from self–organizing teams.
6. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

## 3.7 Extreme Programming (XP)

The most widely used agile process, originally proposed by Kent Beckin 2004. It uses an object-oriented approach.

### 3.7.1 XP Planning

- Begins with the listening, leads to creation of “user stories” that describes required output, features, and functionality. Customer assigns a value (i.e., a priority) to each story.

- Agile team assesses each story and assigns a cost (development weeks. If more than 3 weeks, customer asked to split into smaller stories)
- Working together, stories are grouped for a deliverable increment next release.
- A commitment (includes delivery date and other project matters) is made.
- Three ways to plan:

  1. either all commitment will be implemented in a few weeks,
  2. high priority commitment first, or
  3. the riskiest commitment will be implemented first.

## 3.7.2 XP Design (occurs both before and after coding as refactoring is encouraged)

- Follows the KIS principle (keep it simple) Nothing more nothing less than the story.
- Encourage the use of CRC (class-responsibility-collaborator) cards in an object-oriented context. The only design work product of XP. They identify and organize the classes that are relevant to the current software increment.
- For difficult design problems, suggests the creation of “spike solutions” a design prototype for that portion is implemented and evaluated.
- Encourages “refactoring” an iterative refinement of the internal program design. Does not alter the external behavior yet improve the internal structure. Minimize chances of bugs. More efficient, easy to read.

### 3.7.3 XP Coding

- Recommends the construction of a unit test for a story before coding commences. So implementer can focus on what must be implemented to pass the test.
- Encourages “pair programming”. Two people work together at one workstation. Real time problem solving, real time review for quality assurance. Take slightly different roles.

### 3.7.4 XP Testing

- All unit tests are executed daily and ideally should be automated. Regression tests are conducted to test current and previous components.
- "Acceptance tests" are defined by the customer and executed to assess customer visible functionality.

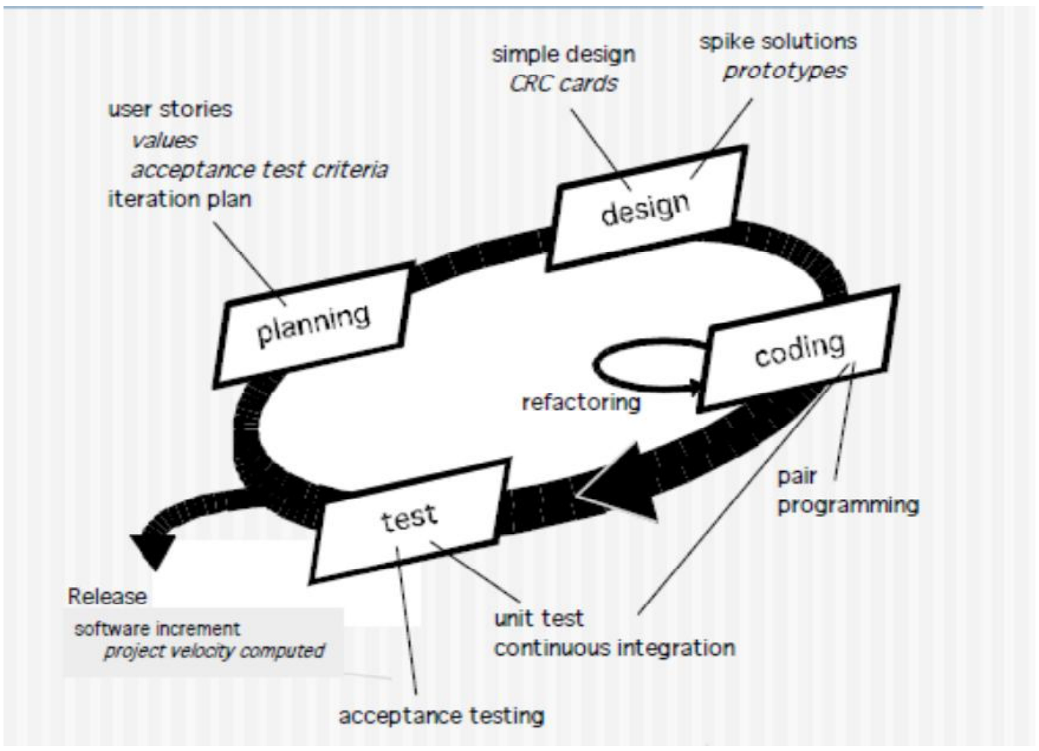


**Figure 3.1 Extreme Programming (XP)**

## 3.7.5 The XP Debate

- **Requirements volatility:** customer is an active member of XP team, changes to requirements are requested informally and frequently.
- **Conflicting customer needs:** Different customers' needs, need to be assimilated.

Different vision or beyond their authority.

- **Requirements are expressed informally:** Use Commitment and acceptance tests are the only explicit.

- Formal models may avoid inconsistencies and errors before the system is built. Proponents said changing nature makes such models obsolete as soon as they are developed.
- **Lack of formal design:** XP deemphasizes the need for architectural design. Complex systems need overall structure to exhibit quality and maintainability. Proponents said incremental nature limits complexity as simplicity is a core value.

## 3.8 Adaptive Software Development (ASD)

- Originally proposed by Jim Highsmith (2000) focusing on human collaboration and team self-organization as a technique to build complex software and system.
- ASD—distinguishing features
  - Mission-driven planning
  - Component-based focus
  - Uses "time-boxing"
  - Explicit consideration of risks
  - Emphasizes collaboration for requirements gathering
  - Emphasizes "learning" throughout the process

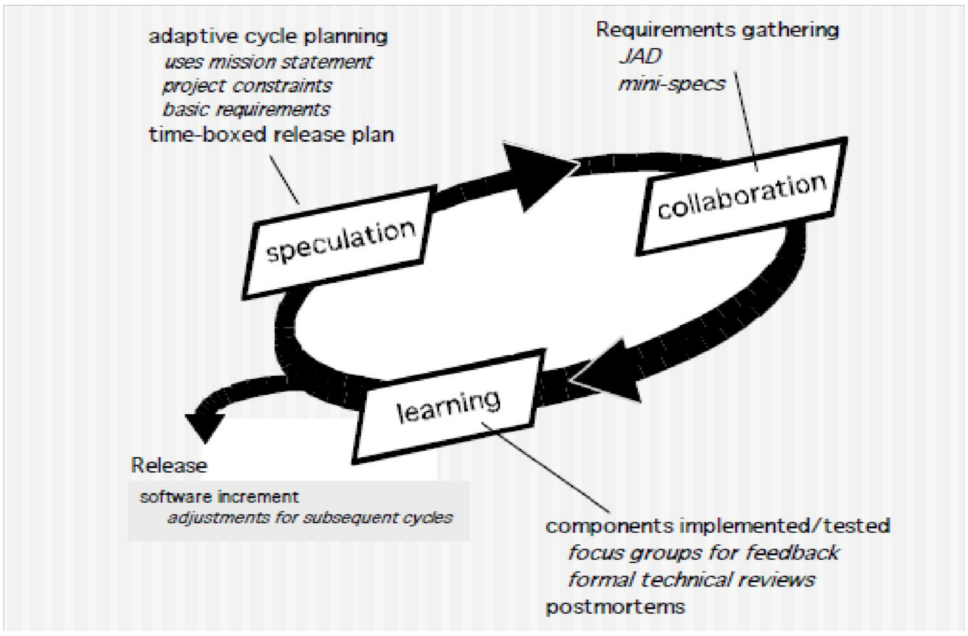


**Figure 3.3. Adaptive Software Development**

### 3.8.1 Three Phases of ASD

**Speculation:**

Project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning uses project initiation information- the customer's mission statement, project constraints (e.g. delivery date), and basic requirements to define the set of release cycles (increments) that will be required for the project. Based on the information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality.

**Collaborations**

It is used to multiply their talent and creative output beyond absolute number (1+1>2). It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is a matter of trust.

1) Criticize without animosity,

2) Assist without resentments,

3) work as hard as or harder than they do.

4) have the skill set to contribute to the work at hand,

5) Communicate problems or concerns in a way that leads to effective action.

**Learning:**

As members of ASD team begin to develop the components, the emphasis is on "learning". Highsmith argues that software developers often overestimate their own understanding of the technology, the process, and the project and that learning will help them to improve their level of real understanding. Three ways: focus groups, technical reviews and project postmortems.

## Check Your Progress3.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- Define the term 'Agility'.
- Understand the Agile process and its agile software development strategy.

- Understand about the two principles of Agile viz., Agile Principle I and Agile Principle II in detail.
- Understand about the different software used in agile methodology namely, Extreme Programming (XP), Adaptive Software Development (ASD), Dynamic Systems Development Method1**. Fill up the blanks**

___________ means effective (rapid and adaptive) response to change.

2. Extreme Programming uses ______________ approach

3**. Choose the correct answer**

Adaptive Software Development (ASD) uses

a. time-boxing b. decision

c. pair programming d. testing

4. XP deemphasizes the need for ______

a. structural design b. architectural design

c. detailed design d. coding

5. Say True or False

Based on Agility Principle II, Working software is the primary measure of progress.

## 3.9 Dynamic Systems Development Method

- It is an agile software development approach that provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment.
- Promoted by the DSDM Consortium (www.dsdm.org)
- DSDM—distinguishing features
    - Similar in most respects to XP and/or ASD
    - Nine guiding principles

- Active user involvement is imperative.
- DSDM teams must be empowered to make decisions.
- The focus is on frequent delivery of products.
- Fitness for business purpose is the essential criterion for acceptance of deliverables.
- Iterative and incremental development is necessary to converge on an accurate business solution.
- All changes during development are reversible.
- Requirements are base lined at a high level
- Testing is integrated throughout the life-cycle

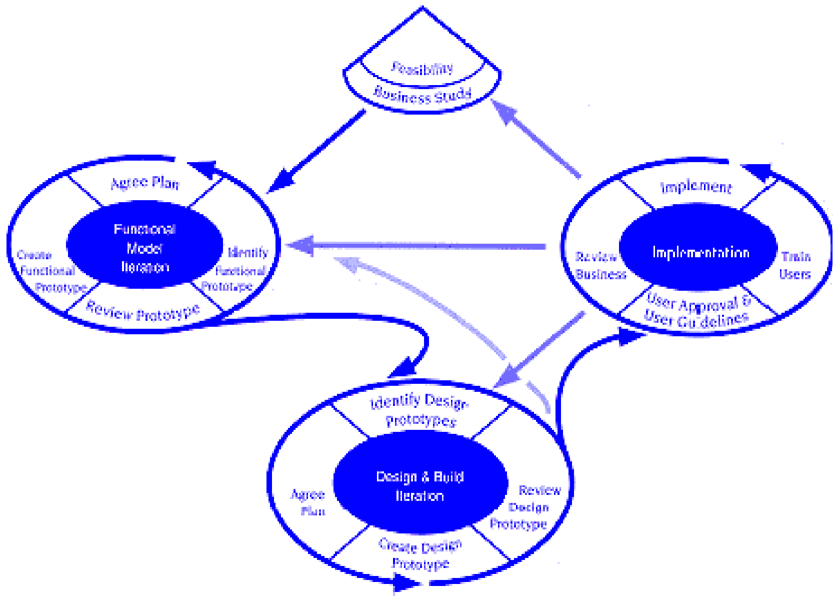


**Figure 3.4 DSDM Life Cycle (with permission of the DSDM consortium)**

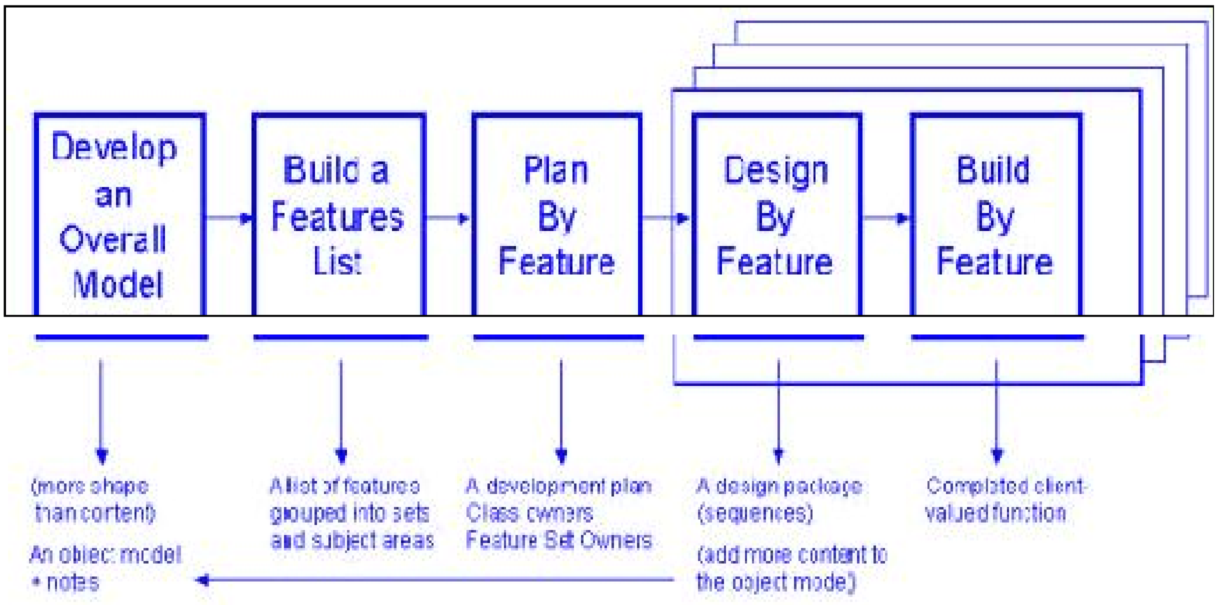
## Crystal

- Proposed by Cockburn and Highsmith
- Actually a family of process models that allow “manoeuvrability” based on problem characteristics

- *Face-to-face communication is emphasized*
- *Suggests the use of "reflection workshops" to review the work habits of the team*

**Feature Driven Development**

- Originally proposed by Peter Coad et al as a object-oriented software engineering process model.
- FDD—distinguishing features
  - o Emphasis is on defining "features" which can be organized hierarchically.
  - o a feature "is a client-valued function that can be implemented in two weeks or less."
  - o Uses a feature template
    - <action> the <result><by | for | of | to> a(n)<object>
    - E.g. Add the product to shopping cart.
    - Display the technical-specifications of the product.



Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
An object model + notes
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
(add more content to the object model)
Completed client-valued function

**Figure 3.5 Feature Driven Development**

- Store the shipping-information for the customer.
- A features list is created and “plan by feature” is conducted Design and construction merge in FDD

**Agile Modeling**

- ***Originally proposed by Scott Ambler***
- ***Suggests a set of agile modeling principles***
  - ***Model with a purpose***
  - ***Use multiple models***
  - ***Travel light***
  - ***Content is more important than representation***
  - ***Know the models and the tools you use to create them***
  - ***Adapt locally***

## 3.10 Summary

Agile Modelling (AM) is a practice-based methodology for effective modelling and documentation of software-based systems. It gives detail description about the two Agility Principles I and II . It explains about Adaptive Software Development (ASD) methods and its three phases . It further explains about the Extreme programming and its steps.

## 3.11 Check your Answers

1. Agility
2. object-oriented
3. time -boxing
4. architectural design
5. True

## 3.12 Model Questions

1. What is Agility?
2. Write briefly about the Extreme Programming.
3. What are the various Agility Principles available?
4. What is Agile Modelling?
5. Write briefly about the Adaptive Software Development (ASD).

# LESSON-4

# REQUIREMENTS ENGINEERING

## 4.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- Understand the term requirement elicitation.
- Understand the distinct steps of requirement engineering viz,., . requirements elicitation, requirements analysis and negotiation, requirements specification, system modeling, requirements validation, requirements management.

## Structure

## 4.2 Introduction

The outcome of the system engineering process is the specification of a computer based system or product at the different levels described generically , But the challenge facing system engineers (and software engineers) is profound: How can we ensure that we have specified a system that properly meets the customer’s needs and satisfies the customer’s expectations? There is no foolproof answer to this difficult question, but a solid requirements engineering process is the best solution we currently have.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. The requirements engineering process can be described in five distinct steps

- requirements elicitation
- requirements analysis and negotiation
- requirements specification
- system modeling
- requirements validation
- requirements management

## 4.3 Requirements Elicitation

Requirements elicitation is a process of identifying needs and bridging the disparities among the involved communities for the purpose of defining and distilling requirements to meet the constraints of these communities. Other often used names for requirements elicitation are requirements acquisition, requirements capture, requirements discovery, requirements gathering, problem analysis, understanding etc. This process requires application domain knowledge, organizational knowledge and technical knowledge, as well as specific problem knowledge. Requirements elicitations are non-trivial because you cannot get all requirements from user and customer by just asking what a system should do.

A lot of techniques can be used in the requirements elicitation process .

Some examples are:

- **Interviews:** This is the most commonly used technique.
- **Ethnography:** This technique is adopted from sociology and includes the observation of users.
- **Questionnaire:** Question is prepared with limited choice of options.
- **Contextual query:** This is a knowledge acquisition approach.

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a dayto-day basis. But it isn't simple—it's very hard.

Christel and Kang [CRI92] identify a number of problems that help us understand why requirements elicitation is difficult.

• **Problems of scope**

The boundary of the system is ill-defined or the customers/ users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

• **Problems of understanding**

The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.

• **Problems of volatility**

The requirements change over time. To help overcome these problems, system engineers must approach the requirements gathering activity in an organized manner.

Sommerville and Sawyer [SOM97] suggest a set of detailed guidelines for requirements elicitation, which are summarized in the following steps:

• Assess the business and technical feasibility for the proposed system.

• Identify the people who will help specify requirements and understand their organizational bias.

• Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.

• Identify "domain constraints" (i.e., characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to be built.

• Define one or more requirements elicitation methods (e.g., interviews, focus groups, team meetings).

• Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded.

• Identify ambiguous requirements as candidates for prototyping.

• Create usage scenarios to help customers/users better identify key requirements. The work products produced as a consequence of the requirements elicitation activity will vary depending on the size of the system or product to be built. For most systems,

the work products include

• A statement of need and feasibility.

• A bounded statement of scope for the system or product.

• A list of customers, users, and other stakeholders who participated in the requirements elicitation activity.

• A description of the system's technical environment.

• A list of requirements (preferably organized by function) and the domain constraints that apply to each.

• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.

• Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in the requirements elicitation.

## 4.4 Requirements Analysis and Negotiation

Requirements Analysis and Negotiation is a process during which requirements are analyzed and modeled. Possible conflicts are resolved by negotiation between stakeholders. The elicitation process provides the input to this process. The output of the process is a consistent and complete set of requirements. Some typical techniques that can be used during this phase are:

- **Unified Modeling Language (UML)**: UML is a collection of techniques for modeling software or systems.
- **Specification and Description Language (SDL)**: SDL is a relatively mature requirement description and definition language.
- **Structured Analysis Structured Design (SASD)**: SASD uses a set of different techniques such as functional decomposition technique, data-flow diagram and data dictionary.
- **Goal-based techniques**: These techniques can be used in the later stage of requirements elicitation as well as very effective requirements analysis techniques.
- **Petri Nets**: Petri Nets are good at modeling state transitions and control flow in an asynchronous system where there is a lot of parallel and asynchronous events.

Once requirements have been gathered, the work products noted earlier form the basis for requirements analysis. Analysis categorizes requirements and organizes them into related subsets; explores each requirement in relationship to others; examines requirements for consistency, omissions, and ambiguity; and ranks requirements based on the needs of customers/users.

As the requirements analysis activity commences, the following questions are asked and answered:

• Is each requirement consistent with the overall objective for the system/product?

• Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?

• Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?

• Is each requirement bounded and unambiguous?

• Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?

• Do any requirements conflict with other requirements?

• Is each requirement achievable in the technical environment that will house the system or product?

• Is each requirement testable, once implemented?

It isn’t unusual for customers and users to ask for more than can be achieved, given limited business resources. It also is relatively common for different customers or users to propose conflicting requirements, arguing that their version is “essential for our special needs.”

## 4.5 Requirements Specification

The Requirements document or Software Requirements Specification describes external behavior of software system. This document can be written by user/customer or developer. A well written requirements document helps in meeting the desired goals of successful software within time limit. Errors that are found in the requirements document are easy to fix, cost less and consume less time. If these documents are handled properly errors can be reduced. There are two types of errors that can be found in requirements document. Knowledge errors, which are caused due to not knowing what the requirements are and Specification errors, caused due to lack of knowledge or experience of specifying requirements.

**INTERNAL** attributes of requirements documents describe how requirements should be specified. What they should include and how they affect others attributes.

**Unambiguous**

A requirement document should be unambiguous but natural language such as English has many inherent ambiguous words. Deterministic Finite State Machine (FM), Pertinens, and Decisional Tree are used for formal specification because they have less ambiguity inherent in them.

**Correct**

A requirements document is correct if and only if every sentence or requirement mentioned in the document is built by the system.

**Complete**

A requirement document is said to be complete if it satisfies 4 conditions:

1. The document should include each and everything that the software supposed to do .

2. All pages, tables and figures are numbered. Names and references should be present in referenced material.

3. In the document no “TB” To Be Determined.

There should be some output for every input of the system.

**Understandable**

A requirement document should be understandable by all readers which include customers, users, project managers etc.

**Verifiable**

A requirements document is verifiable if there exist finite cost effective process with which a person or machine can check that the built software product meets the requirements.

**Internal Consistent**

A requirements document is said to be internally consistent if no subset of requirements conflict such as focus, drag and press with other requirements of a set.

**Modifiable:**

A requirements document is said to be modifiable if changes can be made consistently. Improvements and removing errors are also important for modification of the documents.

**Annotated by Relative Importance:**

A requirements documents is said to be annotated by relative importance if its features are in ascending order according to there importance to the customers. They can be marked as compulsory, recommended and optional .

**Annotated by Relative Stability:**

A requirements documents can be annotated by relative stability depending on the help by designers. Changes of requirements is the cause of the stability of the features like which can be prone to early changes or which features can stay for longer period.

**Annotated by Version :**

A requirements documents is said to be annotated by version if the reader can determine which features can be implemented in which version of the software. This attribute can be application dependent.

**Precise :**

A requirements document is precise if it should not contain vague details. For example the system should not keep the user waiting for acknowledgment but it should have certain time like 2 minutes.

**Traced:**

A requirements document is traced if the origin of each of its requirements have references to earlier versions of the supportive documents.

**Traceable:**

A requirements document is said to be traceable if it has reference of every requirement stated . This can be achieved by numbering every paragraphs and requirements uniquely.

**Not Redundant:**

If the requirements document has same information stored in more than one place it is called redundant. Redundancy is helpful for readability but it can also create problems. Change at one place not at the other places leads to inconsistencies.

**At right level of detail :**

A requirements document provides different levels of details. It can be from very general to very specific. It should cover the minimum functions specified by the customer. For example, System should accept payment, cash payment and previous bills with credit cards.

**Organized:**

A requirements document is organized if its contents are made for easy navigation of information to users. Grouping the functional requirements by user class, object and features are some methods that make the document organized.

1. **EXTERNAL** attributes of requirements documents describe overall or outer appearance of SRS.

**Achievable :**

A requirements document is achievable if there exists at least one system design and implementation that correctly implements all the requirements stated in the document. This can be done by building prototypes of the system .

**Electronically Stored:**

If a word processor is used for storing the requirements documents then it is electronically stored .It can be generated from requirements database.

**Concise:** A requirements document is said to be concise if the length of the document is made short without effecting other qualities.

**Design Independent:**

Requirements document should be design independent. No single solution should be made compulsory if there are many ways to solve a problem. Only required features should be mentioned instead of imposing a design.

**Reusable:**

A requirements document is said to be reusable when it is used for the future requirements documents which is helpful in reducing time. Reusability could be of general terms in non-similar projects and specialized for similar projects.

In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Some suggest that a “standard template” [SOM97] should be developed and used for a system specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

The System Specification is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element.

The System Specification also describes the information (data and control) that is input to and output from the system.

## 4.6 System Modeling

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification.

Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation. You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.

2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

The most important aspect of a system model is that it leaves out detail. A model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a representation of a system should maintain all the information about the entity being represented but unfortunately, the real world (also knows as the universe of discourse) is utterly complex so weed to simplify. An abstraction consciously simplifies and picks out the most evident characteristics.

You may develop different models to represent the system from different perspectives. For example:

1. An external perspective, where you model the context or environment of the system.

2. An interaction perspective where you model the interactions between a system and its environment or between the components of a system.

3. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.

4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

These perspectives have much in common with Krutchen's 4 + 1 view of system architecture, where he suggests that you should document a system's architecture and organization from different perspectives.

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. The detail and rigor of a model depends on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a means of facilitating discussion about an existing or proposed system.
2. As a way of documenting an existing system.
3. As a detailed system description that can be used to generate a system implementation.

In the first case, the purpose of the model is to stimulate the discussion amongst the software engineers involved in developing the system. The models may be incomplete (so long as they cover the key points of the discussion) and they may use the modeling notation informally. This is how models are normally used in so-called 'agile modeling'.

When models are used as documentation, they do not have to be complete as you may only wish to develop models for some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.

In the third case, where models are used as part of a model-based development process, the system models have to be both complete and correct. The reason for this is that they are used as a basis for generating the source code of the system. Therefore, you have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that have different meanings.

Assume for a moment that you have been asked to specify all requirements for the construction of a gourmet kitchen. You know the dimensions of the room, the location of doors and windows, and the available wall space. You could specify all cabinets and appliances and even indicate where they are to reside in the kitchen. Would this be a useful specification?

The answer is obvious. In order to fully specify what is to be built, you would need a meaningful model of the kitchen, that is, a blueprint or three-dimensional rendering that shows the position of the cabinets and appliances and their relationship to one another. From the model, it would be relatively easy to assess the efficiency of work flow (a requirement for all kitchens), the aesthetic “look” of the room (a personal, but very important requirement).

We build system models for much the same reason that we would develop a blueprint or 3D rendering for the kitchen. It is important to evaluate the system’s components in relationship to one another, to determine how requirements fit into this picture, and to assess the “aesthetics” of the system as it has been conceived.

## 4.7 Requirements Validation

It’s a process of ensuring the specified requirements meet the customer needs. It’s concerned with finding problems with the requirements. These problems can lead to extensive rework costs when these they are discovered in the later stages, or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or code errors. Because a change to the requirements usually means the design and implementation must also be changed, and re-tested.

During the requirements validation process, different types of checks should be carried out on the requirements. These checks include:

1. **Validity** *checks*: The functions proposed by stakeholders should be aligned with what the system needs to perform. You may find later that there are additional or different functions are required instead.

2. **Consistency** *checks*: Requirements in the document shouldn’t conflict or different description of the same function

3. **Completeness** *checks*: The document should include all the requirements and constrains.

4. **Realism** *checks*: Ensure the requirements can actually be implemented using the knowledge of existing technology, the budget, schedule, etc.

5. **Verifiability**: Requirements should be written so that they can be tested. This means you should be able to write a set of tests that demonstrate that the system meets the specified requirements.

The work products produced as a consequence of requirements engineering (a system specification and related information) are assessed for quality during a validation step.

Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the formal technical review. The review team includes system engineers, customers, users, and other stakeholders who examine the system specifications looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements. Although the requirements validation review can be conducted in any manner that results in the discovery of requirements errors, it is useful to examine each requirement against a set of checklist questions. The following questions represent a small subset of those that might be asked:

• Are requirements stated clearly? Can they be misinterpreted?

• Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?

• Is the requirement bounded in quantitative terms?

• What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?

• Does the requirement violate any domain constraints?

• Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?

• Is the requirement traceable to any system model that has been created?

• Is the requirement traceable to overall system/product objectives?

• Is the system specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?

• Has an index for the specification been created?

• Have requirements associated with system performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

The development of software begins once the requirements document is ‘ready’. One of the objectives of this document is to check whether the delivered software system is acceptable. For this, it is necessary to ensure that the requirements specification contains no errors and that it specifies the user’s requirements correctly. Also, errors present in the SRS will adversely affect the cost if they are detected later in the development process or when the software is delivered to the user. Hence, it is desirable to detect errors in the requirements before the design and development of the software begins. To check all the issues related to requirements, requirements validation is performed.

In the validation phase, the work products produced as a consequence of requirements engineering are examined for consistency, omissions, and ambiguity. The basic objective is to ensure that the SRS reflects the actual requirements accurately and clearly. Other objectives of the requirements document are listed below.

1. To certify that the SRS contains an acceptable description of the system to be implemented
2. To ensure that the actual requirements of the system are reflected in the SRS
3. To check the requirements document for completeness, accuracy, consistency, requirement conflict’, conformance to standards and technical errors.

Requirements validation is similar to requirements analysis as both processes review the gathered requirements. Requirements validation studies the ‘final draft’ of the requirements document while requirements analysis studies the ‘raw requirements’ from the system stakeholders (users). Requirements validation and requirements analysis can be summarized as follows:

1. **Requirements validation:** Have we got the requirements right?
2. **Requirements analysis:** Have we got the right requirements?

Various inputs such as requirements document, organizational knowledge, and organizational standards are shown. The requirements document should be formulated and organized according to the standards of the organization. The **organizational knowledge** is used to estimate the realism of the requirements of the system. The **organizational standards are** specified standards followed by the organization according to which the system is to be developed.

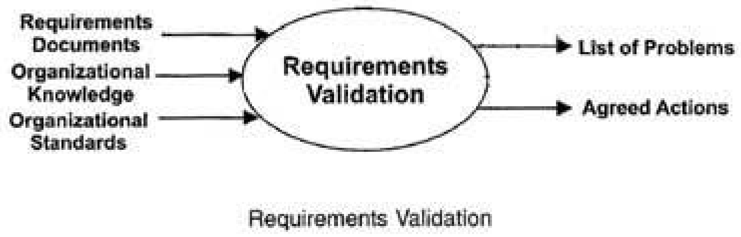


**Figure 4.1 Requirements Validation**

The output of requirements validation is a list of problems and agreed actions of the problems. The **lists of problems** indicate the problems encountered in the" requirements document of the requirements validation process. The **agreed actions** is a list that displays the actions to be performed to resolve the problems depicted in the problem list.

**Requirements validation** is the process of checking that requirements defined for development, define the system that the customer really wants. To check issues related to requirements, we perform requirements validation. We usually use requirements validation to check error at the initial phase of development as the error may increase excessive rework when detected later in the development process.

In the r equirements validation process, we perform a different type of test to check the requirements mentioned in the Software Requirements Specification (SRS), these checks include:

- Completeness checks
- Consistency checks
- Validity checks

- Realism checks
- Ambiguity checks
- Verifiability

The output of requirements validation is the list of problems and agreed on actions of detected problems. The lists of problems indicate the problem detected during the process of requirement validation. The list of agreed action states the corrective action that should be taken to fix the detected problem.

There are several techniques which are used either individually or in conjunction with other techniques to check to check entire or part of the system:

**Test case generation:**

Requirement mentioned in SRS document should be testable, the conducted tests reveal the error present in the requirement. It is generally believed that if the test is difficult or impossible to design than, this usually means that requirement will be difficult to implement and it should be reconsidered.

1. **Prototyping:**

   In this validation techniques the prototype of the system is presented before the end-user or customer, they experiment with the presented model and check if it meets their need. This type of model is generally used to collect feedback about the requirement of the user.

2. **Requirements Reviews:**

   In this approach, the SRS is carefully reviewed by a group of people including people from both the contractor organizations and the client side, the reviewer systematically analyses the document to check error and ambiguity.

3. **Automated Consistency Analysis:**

   This approach is used for automatic detection of an error, such as nondeterminism, missing cases, a type error, and circular definitions, in requirements specifications.

First, the requirement is structured in formal notation then CASE tool is used to check in-consistency of the system, The report of all inconsistencies is identified and corrective actions are taken.

4. **Walk-through:**

A walkthrough does not have a formally defined procedure and does not require a differentiated role assignment.

- o Checking early whether the idea is feasible or not.
- o Obtaining the opinions and suggestion of other people.
- o Checking the approval of others and reaching an agreement.

### Check your Progress

1. **Fill up the blanks**System modeling is the process of developing__________ models of a system.
2. Software Requirements Specification describes ____________ behavior of software system.
3. **Choose the correct answer:**System modeling is the process of developing ____________models of a system.

   a. Constructive b . abstract c. design d. destructive
4. SDL is a relatively mature requirement description and ________language.

   a. Non structured b. procedural c. definition d. structured
5. Say True or FalseIn requirements validation process, we perform a different type of test.

## 4.8 Requirements Management

Requirements Management is a process of managing changes to requirements. Changes are inevitable because of system errors and better understanding development of customers real needs. Requirements Management is carried out in parallel with other requirements activities in the software development.

Activities of requirements management includes keeping project plan undated with requirements, controlling requirements versions, tracking status of requirements and tracing the requirements. Requirements management is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds.

Like SCM, requirements management begins with identification. Each requirement is assigned a unique identifier that might take the form

<requirement type><requirement #>

where requirement type takes on values such as

F = functional requirement,

D = data requirement,

B = behavioral requirement,

I = interface requirement, and P = output requirement.

Hence, a requirement identified as F09 indicates a functional requirement assigned requirement number 9.

Once requirements have been identified, traceability tables are developed. Each traceability table relates identified requirements to one or more aspects of the system or its environment. Among many possible traceability tables are the following:

**Features traceability table:**

Shows how requirements relate to important customer observable system/product features.

**Source traceability table**:

Identifies the source of each requirement.

**Dependency traceability table:**

Indicates how requirements are related to one another.

**Subsystem traceability table:**

Categorizes requirements by the subsystem(s) that they govern.

**Interface traceability table**:

It shows how requirements relate to both internal and external system interfaces. In many cases, these traceability tables are maintained as part of a requirements database so that they may be quickly searched to understand how a change in one requirement will affect different aspects of the system to be built.

## 4.9 Summary

In this lesson it explains about the steps involved in the requirement engineering process. It also explains about the SRS process and tools available are discussed .

## 4.10 Check your answers

**1.** Fill up the blanks: abstract

**2.** external

**3.** abstract

**4.** definition

**5.** True

## 4.11 Model Questions

1. What is Requirement elicitation?

2. What is SRS?

3.Write briefly about the Requirement validation.

4. What is Walkthrough?

5. Write briefly about system modeling.

# LESSON-5

# DATA MODELLING

## 5.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- define the term ‘modelling’ and know about types of data modelling
- understand the different approaches used in the software engineering; and
- understand the modeling principles and analysis of model.

## Structure

## 5.2 Introduction

In this lesson, Data modelling has been focused. It discuss about the types of models which is used for designing the software. It further discuss about the analysis of models which is essential in constructing models. Software Engineering Methods are explains about their tools and few methods like Heuristic Methods, Formal Methods, Prototyping Methods, and Agile Methods are discussed in detail.

## 5. 3 Data Modelling Approaches

Data modeling is the process of creating a data model for the data to be stored in a Database. This data model is a conceptual representation of

- Data objects
- The associations between different data objects
- The rules.

Data modeling helps in the visual representation of data and enforces business rules, regulatory compliances, and government policies on the data. Data Models ensure consistency in naming conventions, default values, semantics, security while ensuring quality of the data.

Data model emphasizes on what data is needed and how it should be organized instead of what operations need to be performed on the data. Data Model is like architect's building plan which helps to build a conceptual model and set the relationship between data items.

The two types of Data Models techniques are

1. Entity Relationship (E-R) Model
2. UML (Unified Modelling Language)

**Why use Data Model?**

The primary goal of using data model are:

- Ensures that all data objects required by the database are accurately represented. Omission of data will lead to creation of faulty reports and produce incorrect results.
- A data model helps design the database at the conceptual, physical and logical levels.
- Data Model structure helps to define the relational tables, primary and foreign keys and stored procedures.
- It provides a clear picture of the base data and can be used by database developers to create a physical database.
- It is also helpful to identify missing and redundant data.
- Though the initial creation of data model is labor and time consuming, in the long run, it makes your IT infrastructure upgrade and maintenance cheaper and faster.

## 5.4 Types of Data Models

**There are mainly three different types of data models**

1. **Conceptual:** This Data Model defines **WHAT** the system contains. This model is typically created by Business stakeholders and Data Architects. The purpose is to organize, scope and define business concepts and rules.
2. **Logical:** Defines **HOW** the system should be implemented regardless of the DBMS. This model is typically created by Data Architects and Business Analysts. The purpose is to developed technical map of rules and data structures.
3. **Physical:** This Data Model describes **HOW** the system will be implemented using a specific DBMS system. This model is typically created by DBA and developers. The purpose is actual implementation of the database.

**Conceptual Model**

The main aim of this model is to establish the entities, their attributes, and their relationships. In this Data modeling level, there is hardly any detail available of the actual Database structure.

The 3 basic tenants of Data Model are

**Entity**: A real-world thing

**Attribute**: Characteristics or properties of an entity

**Relationship**: Dependency or association between two entities

For example:

- Customer and Product are two entities. Customer number and name are attributes of the Customer entity
- Product name and price are attributes of product entity
- Sale is the relationship between the customer and product

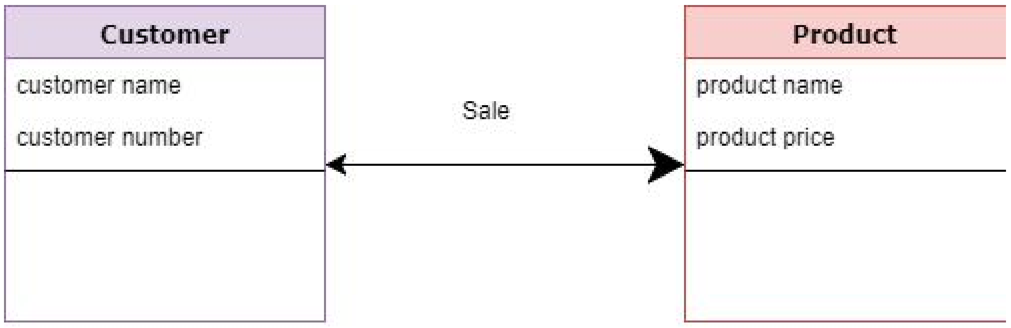


**Figure 5.1 – Conceptual Model for Customer and Product relation**

**Characteristics of a conceptual data model**

- Offers Organisation-wide coverage of the business concepts.
- This type of Data Models are designed and developed for a business audience.
- The conceptual model is developed independently of hardware specifications like data storage capacity, location or software specifications like DBMS vendor and technology. The focus is to represent data as a user will see it in the "real world."

Conceptual data models known as Domain models create a common vocabulary for all stakeholders by establishing basic concepts and scope.

**Logical Data Model**

Logical data models add further information to the conceptual model elements. It defines the structure of the data elements and set the relationships between them.

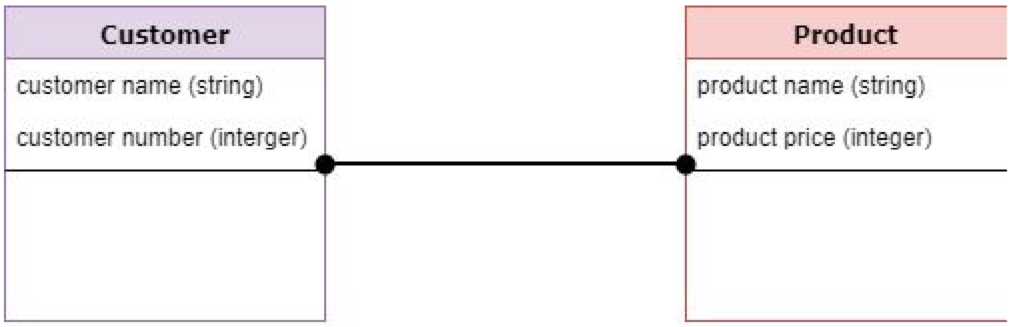


**Figure 5.2 – Logical Data Model for Customer and Product relation**

The advantage of the Logical data model is to provide a foundation to form the base for the Physical model. However, the modeling structure remains generic.

At this Data Modeling level, no primary or secondary key is defined. At this Data modeling level, you need to verify and adjust the connector details that were set earlier for relationships.

**Characteristics of a Logical data model**

- Describes data needs for a single project but could integrate with other logical data models based on the scope of the project.
- Designed and developed independently from the DBMS.
- Data attributes will have datatypes with exact precisions and length.
- Normalization processes to the model is applied typically till 3NF.

**Physical Data Model**

A Physical Data Model describes the database specific implementation of the data model. It offers an abstraction of the database and helps generate schema. This is because of the richness of meta-data offered by a Physical Data Model.

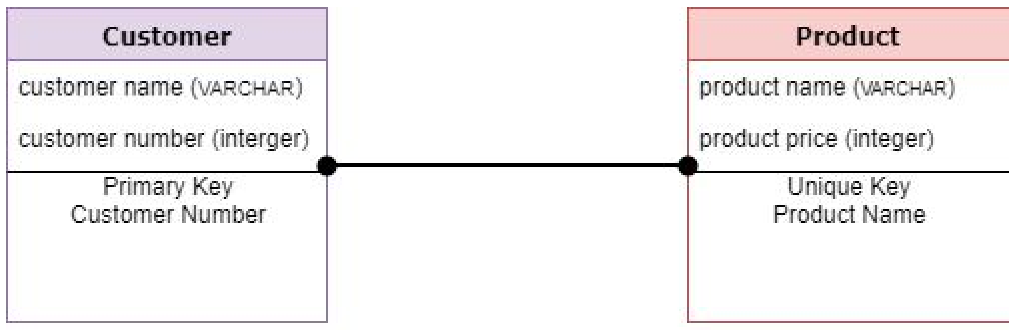


**Figure 5.3 – Physical Data Model for Customer and Product relation**

This type of Data model also helps to visualize database structure. It helps to model database columns keys, constraints, indexes, triggers, and other RDBMS features.

**Characteristics of a physical data model**

- The physical data model describes data need for a single project or application though it may be integrated with other physical data models based on project scope.
- Data Model contains relationships between tables that which addresses cardinality and nullability of the relationships.
- Developed for a specific version of a DBMS, location, data storage or technology to be used in the project.
- Columns should have exact datatypes, lengths assigned and default values.
- Primary and Foreign keys, views, indexes, access profiles, and authorizations, etc. are defined.

**Advantages of Data model**

- The main goal of a designing data model is to make certain that data objects offered by the functional team are represented accurately.
- The data model should be detailed enough to be used for building the physical database.
- The information in the data model can be used for defining the relationship between tables, primary and foreign keys, and stored procedures.
- Data Model helps business to communicate the within and across organizations.
- Data model helps to documents data mappings in ETL process
- Help to recognize correct sources of data to populate the model

**Disadvantages of Data model**

- To develop Data model one should know physical data stored characteristics.
- This is a navigational system produces complex application development, management. Thus, it requires a knowledge of the biographical truth.
- Even smaller change made in structure require modification in the entire application.
- There is no set data manipulation language in DBMS.

## 5.5 Software Engineering Models

Software engineering models and methods impose structure on software engineering with the goal of making that activity systematic, repeatable, and ultimately more success-oriented. Using models provides an approach to problem solving, a notation, and procedures for model construction and analysis.

Methods provide an approach to the systematic specification, design, construction, test, and verification of the end-item software and associated work products. Software engineering models and methods vary widely in scope—from addressing a single software life cycle phase to covering the complete software life cycle.

The emphasis in this knowledge area (KA) is on software engineering models and methods that encompass multiple software life cycle phases, since methods specific for single life cycle phases are covered by other KAs. The breakdown of topics for the Software Engineering Models and Methods KA is shown in Figure 5.4.

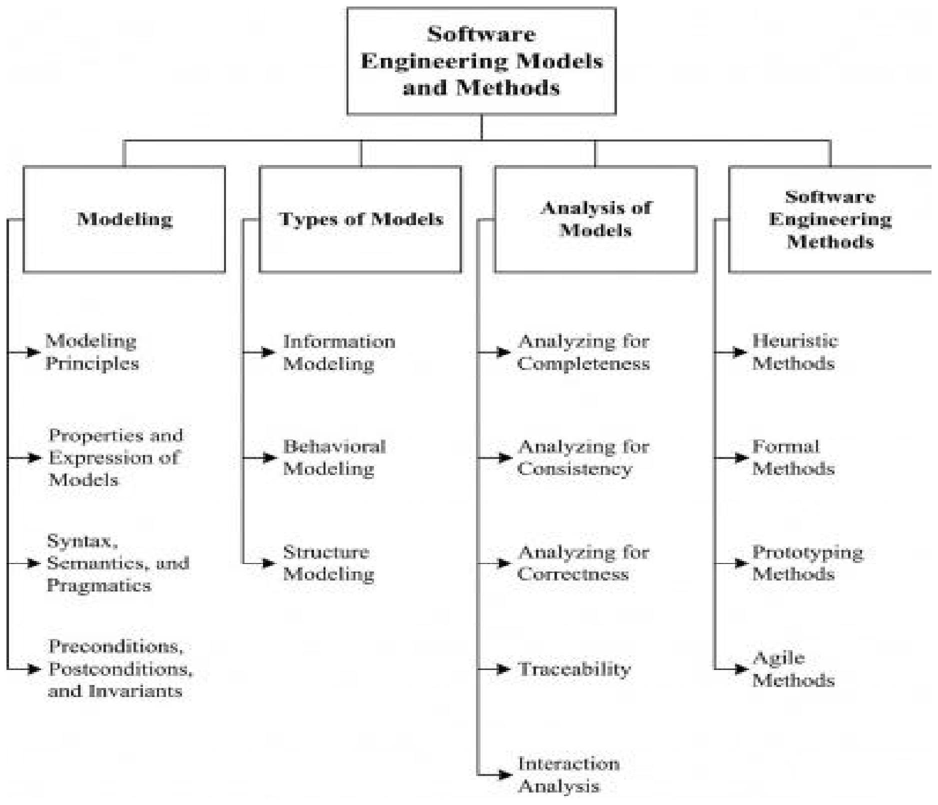

Software Engineering Models and Methods
Modeling
Types of Models
Analysis of Models
Software Engineering Methods
Modeling Principles
Properties and Expression of Models
Syntax, Semantics, and Pragmatics
Preconditions, Postconditions, and Invariants
Information Modeling
Behavioral Modeling
Structure Modeling
Analyzing for Completeness
Analyzing for Consistency
Analyzing for Correctness
Traceability
Interaction Analysis
Heuristic Methods
Formal Methods
Prototyping Methods
Agile Methods

**Figure 5.4: Breakdown of Topics for the Software Engineering Models and Methods KA**

## 5.6 Modeling

Modeling of software is becoming a pervasive technique to help software engineers understand, engineer, and communicate aspects of the software to appropriate stakeholders. Stakeholders are those persons or parties who have a stated or implied interest in the software (for example, user, buyer, supplier, architect, certifying authority, evaluator, developer, software engineer, and perhaps others).

While there are many modeling languages, notations, techniques, and tools in the literature and in practice, there are unifying general concepts that apply in some form to them all. The following sections provide background on these general concepts.

### 5.6.1 Modeling Principles

Modeling provides the software engineer with an organized and systematic approach for representing significant aspects of the software under study, facilitating decision-making about the software or elements of it, and communicating those significant decisions to others in the stakeholder communities. There are three general principles guiding such modeling activities:

- Model the Essentials: good models do not usually represent every aspect or feature of the software under every possible condition.

Modeling typically involves developing only those aspects or features of the software that need specific answers, abstracting away any nonessential information. This approach keeps the models manageable and useful.

- Provide Perspective: modeling provides views of the software under study using a defined set of rules for expression of the

model within each view. This perspective driven approach provides dimensionality to the model (for example, a structural view, behavioral view, temporal view, organizational view, and other views as relevant). Organizing information into views focuses the software modeling efforts on specific concerns relevant to that view using the appropriate notation, vocabulary, methods, and tools.

- Enable Effective Communications: modeling employs the application domain vocabulary of the software, a modeling language, and semantic expression (in other words, meaning within context). When used rigorously and systematically, this modeling results in a reporting approach that facilitates effective communication of software information to project stakeholders.

A model is an abstraction or simplification of a software component. A consequence of using abstraction is that no single abstraction completely describes a software component. Rather, the model of the software is represented as an aggregation of abstractions, which — when taken together — describe only selected aspects, perspectives, or views — only those that are needed to make informed decisions and respond to the reasons for creating the model in the first place.

This simplification leads to a set of assumptions about the context within which the model is placed that should also be captured in the model. Then, when reusing the model, these assumptions can be validated first to establish the relevancy of the reused model within its new use and context.

### 5.6.2 Properties and Expression of Models

Properties of models are those distinguishing features of a particular model used to characterize its completeness, consistency, and correctness within the chosen modeling notation and tooling used. Properties of models include the following:

- Completeness: the degree to which all requirements have been implemented and verified within the model.
- Consistency: the degree to which the model contains no conflicting requirements, assertions, constraints, functions, or component descriptions.
- Correctness: the degree to which the model satisfies its requirements and design specifications and is free of defects.

Models are constructed to represent real-world objects and their behaviors to answer specific questions about how the software is expected to operate. Interrogating the models — either through exploration, simulation, or review may expose areas of uncertainty within the model and the software to which the model refers.

These uncertainties or unanswered questions regarding the requirements, design, and/or implementation can then be handled appropriately. The primary expression element of a model is an entity. An entity may represent concrete artifacts (for example, processors, sensors, or robots) or abstract artifacts (for example, software modules or communication protocols). Model entities are connected to other entities using relations (in other words, lines or textual operators on target entities).

Expression of model entities may be accomplished using textual or graphical modeling languages; both modeling language types connect model entities through specific language constructs.

The meaning of an entity may be represented by its shape, textual attributes, or both. Generally, textual information adheres to language-specific syntactic structure. The precise meanings related to the modeling of context, structure, or behavior using these entities and relations is dependent on the modeling language used, the design rigor applied to the modeling effort, the specific view being constructed, and the entity to which the specific notation element may be attached.

Multiple views of the model may be required to capture the needed semantics of the software. When using models supported with automation, models may be checked for completeness and consistency. The usefulness of these checks depends greatly on the level of semantic and syntactic rigor applied to the modeling effort in addition to explicit tool support. Correctness is typically checked through simulation and/or review.

### 5.6.3 Syntax, Semantics, and Pragmatics

Models can be surprisingly deceptive. The fact that a model is an abstraction with missing information can lead one into a false sense of completely understanding the software from a single model. A complete model (“complete” being relative to the modeling effort) may be a union of multiple sub models and any special function models.

Examination and decision-making relative to a single model within this collection of sub models may be problematic. Understanding the precise meanings of modeling constructs can also be difficult. Modeling languages are defined by syntactic and semantic rules. For textual languages, syntax is defined using a notation grammar that defines valid language constructs (for example, Backus-Naur Form (BNF)).

For graphical languages, syntax is defined using graphical models called metamodels. As with BNF, metamodels define the valid syntactical constructs of a graphical modeling language; the metamodel defines how these constructs can be composed to produce valid models. Semantics for modeling languages specify the meaning attached to the entities and relations captured within the model. For example, a simple diagram of two boxes connected by a line is open to a variety of interpretations.

Knowing that the diagram on which the boxes are placed and connected is an object diagram or an activity diagram can assist in the interpretation of this model. As a practical matter, there is usually a good understanding of the semantics of a specific software model due to the modeling language selected, how that modeling language is used to express entities and relations within that model, the experience base of the modeler(s), and the context within which the modeling has been undertaken and so represented.

Meaning is communicated through the model even in the presence of incomplete information through abstraction; pragmatics explains how meaning is embodied in the model and its context and communicated effectively to other software engineers. There are still instances, however, where caution is needed regarding modeling and semantics.

For example, any model parts imported from another model or library must be examined for semantic assumptions that conflict in the new modeling environment; this may not be obvious. The model should be checked for documented assumptions. While modeling syntax may be identical, the model may mean something quite different in the new environment, which is a different context. Also, consider that as software matures and changes are made, semantic discord can be introduced, leading to errors.

With many software engineers working on a model part over time coupled with tool updates and perhaps new requirements, there are opportunities for portions of the model to represent something different from the original author's intent and initial model context.

### 5.6.4 Preconditions, Postconditions, and Invariants

When modeling functions or methods, the software engineer typically starts with a set of assumptions about the state of the software prior to, during, and after the function or method executes. These assumptions are essential to the correct operation of the function or method and are grouped, for discussion, as a set of preconditions, postconditions, and invariants.

- Preconditions: a set of conditions that must be satisfied prior to execution of the function or method. If these preconditions do not hold prior to execution of the function or method, the function or method may produce erroneous results.

- Postconditions: a set of conditions that is guaranteed to be true after the function or method has executed successfully. Typically, the postconditions represent how the state of the software has changed, how parameters passed to the function or method have changed, how data values have changed, or how the return value has been affected.
- Invariants: a set of conditions within the operational environment that persist (in other words, do not change) before and after execution of the function or method. These invariants are relevant and necessary to the software and the correct operation of the function or method.

## 5.7 Types of Models

A typical model consists of an aggregation of sub models. Each sub model is a partial description and is created for a specific purpose; it may be comprised of one or more diagrams.

The collection of sub models may employ multiple modeling languages or a single modeling language. The Unified Modeling Language (UML) recognizes a rich collection of modeling diagrams. Use of these diagrams, along with the modeling language constructs, brings about three broad model types commonly used: information models, behavioral models, and structure models (see section 1.1).

### 5.7.1 Information Modeling

Information models provide a central focus on data and information. An information model is an abstract representation that identifies and defines a set of concepts, properties, relations, and constraints on data entities.

The semantic or conceptual information model is often used to provide some formalism and context to the software being modeled as viewed from the problem perspective, without concern for how this model is actually mapped to the implementation of the software.

The semantic or conceptual information model is an abstraction and, as such, includes only the concepts, properties, relations, and constraints needed to conceptualize the real-world view of the information. Subsequent transformations of the semantic or conceptual information model lead to the elaboration of logical and then physical data models as implemented in the software.

### 5.7.2 Behavioural Modeling

Behavioral models identify and define the functions of the software being modeled. Behavioral models generally take three basic forms: state machines, control-flow models, and dataflow models. State machines provide a model of the software as a collection of defined states, events, and transitions.

The software transitions from one state to the next by way of a guarded or unguarded triggering event that occurs in the modeled environment. Control-flow models depict how a sequence of events causes processes to be activated or deactivated. Data-flow behavior is typified as a sequence of steps where data moves through processes toward data stores or data sinks.

### 5.7.3 Structure Modeling

Structure models illustrate the physical or logical composition of software from its various component parts. Structure modeling establishes the defined boundary between the software being implemented or modeled and the environment in which it is to operate.

Some common structural constructs used in structure modeling are composition, decomposition, generalization, and specialization of entities; identification of relevant relations and cardinality between entities; and the definition of process or functional interfaces. Structure diagrams provided by the UML for structure modeling include class, component, object, deployment, and packaging diagrams.

## 5.8 Analysis of Models

The development of models affords the software engineer an opportunity to study, reason about, and understand the structure, function, operational usage, and assembly considerations associated with software. Analysis of constructed models is needed to ensure that these models are complete, consistent, and correct enough to serve their intended purpose for the stakeholders.

### 5.8.1 Analysing for Completeness

In order to have software that fully meets the needs of the stakeholders, completeness is critical—from the requirements elicitation process to code implementation. Completeness is the degree to which all of the specified requirements have been implemented and verified.

Models may be checked for completeness by a modeling tool that uses techniques such as structural analysis and state-space reachability analysis (which ensure that all paths in the state models are reached by some set of correct inputs);

Models may also be checked for completeness manually by using inspections or other review techniques (see the Software Quality KA). Errors and warnings generated by these analysis tools and found by inspection or review indicate probable needed corrective actions to ensure completeness of the models.

### 5.8.2 Analysing for Consistency

Consistency is the degree to which models contain no conflicting requirements, assertions, constraints, functions, or component descriptions. Typically, consistency checking is accomplished with the modeling tool using an automated analysis function;

Models may also be checked for consistency manually using inspections or other review techniques (see the Software Quality KA). As with completeness, errors and warnings generated by these analysis tools and found by inspection or review indicate the need for corrective action.

### 5.8.3 Analyzing for Correctness

Correctness is the degree to which a model satisfies its software requirements and software design specifications, is free of defects, and ultimately meets the stakeholders' needs.

Analyzing for correctness includes verifying syntactic correctness of the model (that is, correct use of the modeling language grammar and constructs) and verifying semantic correctness of the model (that is, use of the modeling language constructs to correctly represent the meaning of that which is being modeled).

To analyze a model for syntactic and semantic correctness, one analyzes it—either automatically (for example, using the modeling tool to check for model syntactic correctness) or manually (using inspections or other review techniques)—searching for possible defects and then removing or repairing the confirmed defects before the software is released for use.

### 5.8.4 Traceability

Developing software typically involves the use, creation, and modification of many work products such as planning documents, process specifications, software requirements, diagrams, designs and pseudo-code, handwritten and tool-generated code, manual and automated test cases and reports, and files and data. These work products may be related through various dependency relationships (for example, uses, implements, and tests). As software is being developed, managed, maintained, or extended, there is a need to map and control these traceability relationships to demonstrate software requirements consistency with the software model (see Requirements Tracing in the Software Requirements KA) and the many work products.

Use of traceability typically improves the management of software work products and software process quality; it also provides assurances to stakeholders that all requirements have been satisfied. Traceability enables change analysis once the software is developed and released, since relationships to software work products can easily be traversed to assess change impact.

Modeling tools typically provide some automated or manual means to specify and manage traceability links between requirements, design, code, and/or test entities as may be represented in the models and other software work products. (For more information on traceability, see the Software Configuration Management KA).

### 5.8.5 Interaction Analysis

Interaction analysis focuses on the communications or control flow relations between entities used to accomplish a specific task or function within the software model. This analysis examines the dynamic behavior of the interactions between different portions of the software model, including other software layers (such as the operating system, middleware, and applications). It may also be important for some software applications to examine interactions between the computer software application and the user interface software.

Some software modeling environments provide simulation facilities to study aspects of the dynamic behavior of modeled software. Stepping through the simulation provides an analysis option for the software engineer to review the interaction design and verify that the different parts of the software work together to provide the intended functions.

## 5.9 Software Engineering Methods

Software engineering methods provide an organized and systematic approach to developing software for a target computer. There are numerous methods from which to choose, and it is important for the software engineer to choose an appropriate method or methods for the software development task at hand; this choice can have a dramatic effect on the success of the software project.

Use of these software engineering methods coupled with people of the right skill set and tools enable the software engineers to visualize the details of the software and ultimately transform the representation into a working set of code and data. Selected software engineering methods are discussed below. The topic areas are organized into discussions of Heuristic Methods, Formal Methods, Prototyping Methods, and Agile Methods.

## 5.9.1 Heuristic Methods

Heuristic methods are those experience-based software engineering methods that have been and are fairly widely practiced in the software industry. This topic area contains three broad discussion categories: structured analysis and design methods, data modeling methods, and object-oriented analysis and design methods.

**Structured Analysis and Design Methods**

The software model is developed primarily from a functional or behavioural viewpoint, starting from a high-level view of the software (including data and control elements) and then progressively decomposing or refining the model components through increasingly detailed designs.

The detailed design eventually converges to very specific details or specifications of the software that must be coded (by hand, automatically generated, or both), built, tested, and verified.

**Data Modeling Methods**

The data model is constructed from the viewpoint of the data or information used. Data tables and relationships define the data models.

This data modeling method is used primarily for defining and analysing data requirements supporting database designs or data repositories typically found in business software, where data is actively managed as a business systems resource or asset.

**Object-Oriented Analysis and Design Method**

The object-oriented model is represented as a collection of objects that encapsulate data and relationships and interact with other objects through methods. Objects may be real-world items or virtual items. The software model is constructed using diagrams to constitute selected views of the software.

Progressive refinement of the software models leads to a detailed design. The detailed design is then either evolved through successive iteration or transformed (using some mechanism) into the implementation view of the model, where the code and packaging approach for eventual software product release and deployment is expressed.

### 5.9.2 Formal Methods

Formal methods are software engineering methods used to specify, develop, and verify the software through application of a rigorous mathematically based notation and language. Through use of a specification language, the software model can be checked for consistency (in other words, lack of ambiguity), completeness, and correctness in a systematic and automated or semi-automated fashion.

This topic is related to the Formal Analysis section in the Software Requirements KA. This section addresses specification languages, program refinement and derivation, formal verification, and logical inference.

**Specification Languages**

Specification languages provide the mathematical basis for a formal method; specification languages are formal, higher level computer languages (in other words, not a classic 3rd Generation Language (3GL) programming language) used during the software specification, requirements analysis, and/ or design stages to describe specific input/ output behaviour. Specification languages are not directly executable languages; they are typically comprised of a notation and syntax, semantics for use of the notation, and a set of allowed relations for objects.

**Program Refinement and Derivation**

Program refinement is the process of crea ting a lower level (or more detailed) specification using a series of transformations. It is through successive transformations that the software engineer derives an executable representation of a program.

Specifications may be refined, adding details until the model can be formulated in a 3GL programming language or in an executable portion of the chosen specification language. This specification refinement is made possible by defining specifications with precise semantic properties; the specifications must set out not only the relationships between entities but also the exact runtime meanings of those relationships and operations.

**Formal Verification**

Model checking is a formal verification method; it typically involves performing a state-space exploration or reachability analysis to demonstrate that the represented software design has or preserves certain model properties of interest.

An example of model checking is an analysis that verifies correct program behaviour under all possible interleaving of event or message arrivals.

The use of formal verification requires a rigorously specified model of the software and its operational environment; this model often takes the form of a finite state machine or other formally defined automaton.

**Logical Inference**

Logical inference is a method of designing software that involves specifying preconditions and postconditions around each significant block of the design, and using mathematical logic developing the proof that those preconditions and postconditions must hold under all inputs.

This provides a way for the software engineer to predict software behaviour without having to execute the software. Some Integrated Development Environments (IDEs) include ways to represent these proofs along with the design or code.

## 5.9.3 Prototyping Methods

Software prototyping is an activity that generally creates incomplete or minimally functional versions of a software application, usually for trying out specific new features, soliciting feedback on software requirements or user interfaces, further exploring software requirements, software design, or implementation options, and/or gaining some other useful insight into the software.

The software engineer selects a prototyping method to understand the least understood aspects or components of the software first; this approach is in contrast with other software engineering methods that usually begin development with the most understood portions first. Typically, the prototyped product does not become the final software product without extensive development rework or refactoring.

This section discusses prototyping styles, targets, and evaluation techniques in brief.

**Prototyping Style**

This addresses the various approaches to developing prototypes. Prototypes can be developed as throwaway code or paper products, as an evolution of a working design, or as an executable specification. Different prototyping life cycle processes are typically used for each style. The style chosen is based on the type of results the project needs, the quality of the results needed, and the urgency of the results.

**Prototyping Target**

The target of the prototype activity is the specific product being served by the prototyping effort. Examples of prototyping targets include a requirements specification, an architectural design element or component, an algorithm, or a human machine user interface.

**Prototyping Evaluation Techniques**

A prototype may be used or evaluated in a number of ways by the software engineer or other project stakeholders, driven primarily by the underlying reasons that led to prototype development in the first place. Prototypes may be evaluated or tested against the actual implemented software or against a target set of requirements (for example, a requirements prototype); the prototype may also serve as a model for a future software development effort (for example, as in a user interface specification).

## Check your Progress

**1.** **Fill in the blanks** Data modeling helps in the visual representation of ______.

2. Syntax is defined using graphical models called _____________.

**3.** **Choose the best answer** Behavioural models identify and define the _________of the software being modelled.

a. Constructor b. destructor c. functions d. classes

4. Structure models illustrate the _____or logical composition of software.

a. Conceptual b. architectural c. physical d. low level

5.S ay True or False Model checking is an Informal verification method.

## 5.9.4 Agile Methods

Agile methods were born in the 1990s from the need to reduce the apparent large overhead associated with heavyweight, plan-based methods used in large-scale software-development projects. Agile methods are considered lightweight methods in that they are characterized by short, iterative development cycles, self-organizing teams, simpler designs, code refactoring, test-driven development, frequent customer involvement, and an emphasis on creating a demonstrable working product with each development cycle. Many agile methods are available in the literature; some of the more popular approaches, which are discussed here in brief, include Rapid Application Development (RAD), Extreme Programming (XP), Scrum, and Feature-Driven Development (FDD).

### RAD

Rapid software development methods are used primarily in data-intensive, business systems application development. The RAD method is enabled with special-purpose database development tools used by software engineers to quickly develop, test, and deploy new or modified business applications.

### XP

This approach uses stories or scenarios for requirements, develops tests first, has direct customer involvement on the team (typically defining acceptance tests), uses pair programming, and provides for continuous code refactoring and integration. Stories are decomposed into tasks, prioritized, estimated, developed, and tested. Each increment of software is tested with automated and manual tests; an increment may be released frequently, such as every couple of weeks or so.

### Scrum

This agile approach is more project management-friendly than the others. The scrum master manages the activities within the project increment; each increment is called a sprint and lasts no more than 30 days. A Product Backlog Item (PBI) list is developed from which tasks are identified, defined, prioritized, and estimated. A working version of the software is tested and released in each increment. Daily scrum meetings ensure work is managed to plan.

**FDD**

This is a model-driven, short, iterative software development approach using a five-phase process:

(1) develop a product model to scope the breadth of the domain,

(2) create the list of needs or features,

(3) build the feature development plan,

(4) develop designs for iteration-specific features,

(5) code, test, and then integrate the features.

FDD is similar to an incremental software development approach; it is also similar to XP, except that code ownership is assigned to individuals rather than the team.

FDD emphasizes an overall architectural approach to the software, which promotes building the feature correctly the first time rather than emphasizing continual refactoring. There are many more variations of agile methods in the literature and in practice. Note that there will always be a place for heavyweight, plan-based software engineering methods as well as places where agile methods shine.

There are new methods arising from combinations of agile and plan-based methods where practitioners are defining new methods that balance the features needed in both heavyweight and lightweight methods based primarily on prevailing organizational business needs. These business needs, as typically represented by some of the project stakeholders, should and do drive the choice in using one software engineering method over another or in constructing a new method from the best features of a combination of software engineering methods.

## 5.10 Summary

The various data modeling  have been read in this lesson. The concept about the data modelling is also clearly described . The various approaches of agile methods like Rapid

Application Development (RAD), Extreme Programming (XP), Scrum, and Feature-Driven Development (FDD). are discussed in  detail.

## 5.11 Check Your Answers

**1.** data

**2.** metamodels.

**3.** functions

**4.** physical

**5.** False

## 5.12 Model Questions

1. What is Data modelling?

2. List the types of Data Models.

3. What are the advantages and disadvantages of data model?

4. rite short notes on formal methods in software engineering.

5. Write briefly about different approaches used in agile methods

# LESSON-6

# FLOW ORIENTED MODELING

## 6.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- To draw the DFD diagram.
- understand the different types of model used for designing the system; and
- understand the different representation of symbols for different models used in Design phase.

### Structure

## 6.2 Introduction

In this lesson, explains about the flow oriented model . This model describes about its elements like  Data flow model, Control flow model, Control Specification and Process Specification. It further discuss about the behavioral model and its process.

## 6.3 Flow oriented Model

In this model shows how data objects are transformed by processing the function. The Flow oriented elements are:

**i. Data flow model**

- It is a graphical technique. It is used to represent information flow.
- The data objects are flowing within the software and transformed by processing the elements.
- The data objects are represented by labeled arrows. Transformation are represented by circles called as bubbles.
- DFD shown in a hierarchical fashion. The DFD is split into different levels. It also called as ‘context level diagram’.

**ii. Control flow model**

- Large class applications require a control flow modeling.
- The application creates control information instated of reports or displays.
- The applications process the information in specified time.
- An event is implemented as a Boolean value.

**For example,** the Boolean values are true or false, on or off, 1 or 0.

**iii. Control Specification**

- A short term for control specification is CS PEC.
- It represents the behaviour of the system.

- The state diagram in CSPEC is a sequential specification of the behaviour.
- The state diagram includes states, transitions, events and activities.
- State diagram shows the transition from one state to another state if a particular event has occurred.

**iv. Process Specification**

- A short term for process specification is PSPEC.
- The process specification is used to describe all flow model processes.
- The content of process specification consists narrative text, Program Design Language(PDL) of the process algorithm, mathematical equations, tables or UML activity diagram.

**Flow oriented** represents how data objects are transformed as they move through the system. A data flow diagram (DFD) is the diagrammatic form that is used to complement UML diagrams. Considered by many to be an 'old school' approach, flow-oriented modeling continues to provide a view of the system that is unique.

The DFD takes an input-process-output insight into system requirements and flow.

Data objects are represented by labeled arrows and transformations are represented by circles (called bubbles).

## 6.4 How to Create a Data Flow Model

The DFD diagram enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system.

**Guidelines**:

1. The level 0 data DFD should depict the software/system as a single bubble.
2. Primary I/O should be carefully noted.
3. Refinement should begin by isolating candidate processes, data objects, and data stores.

4. All arrows and bubbles should be labeled with meaningful names.
5. Information flow continuity must be maintained from level to level.
6. One bubble at one time should be refined.

## 6.5 Data Flow Diagram

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

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

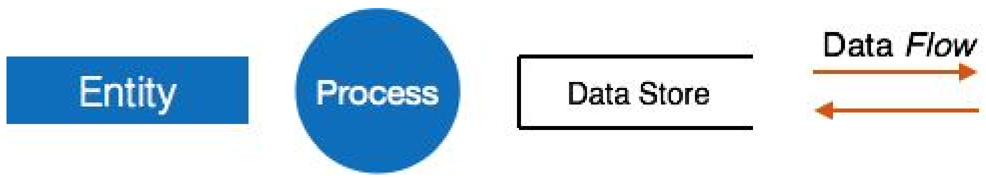
### 6.5.1 Types of DFD

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.

### 6.5.2 DFD Components

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



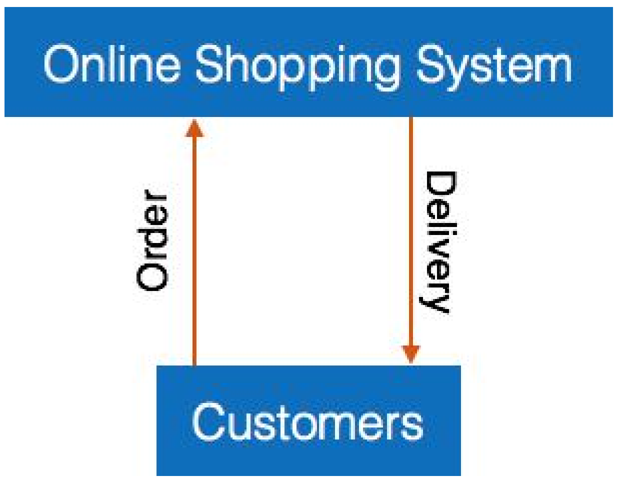
**Symbols used in DFD**

- **Entities** - Entities are source and destination of information data. Entities are represented by a 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.

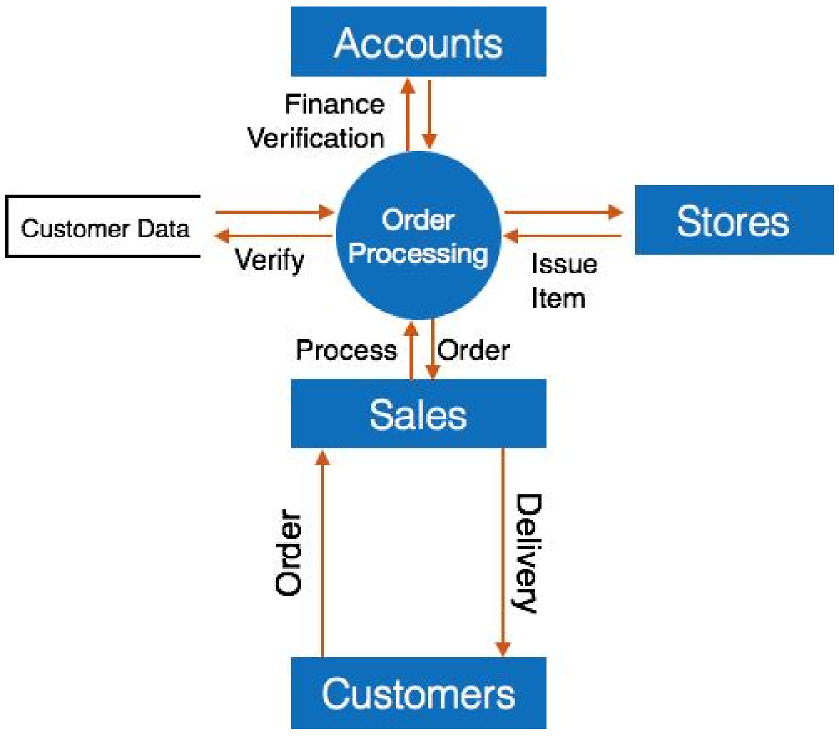
**Levels of DFD**

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



**DFD – Level 0**

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



**DFD – Level 1**

- **Level 2** - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

  Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

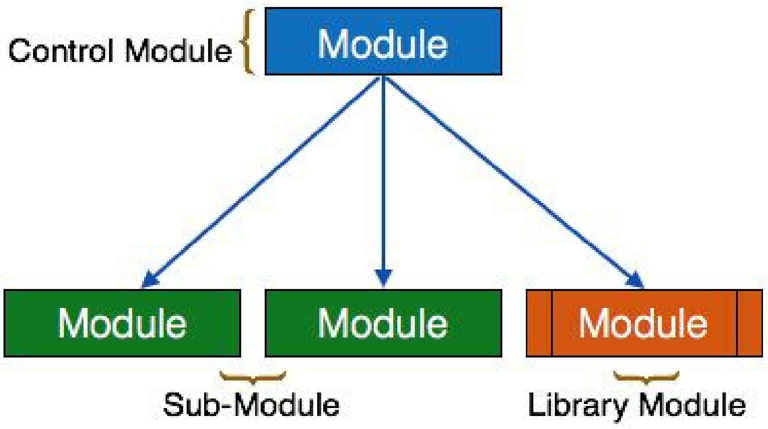
### 6.5.3 Structure Charts

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

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

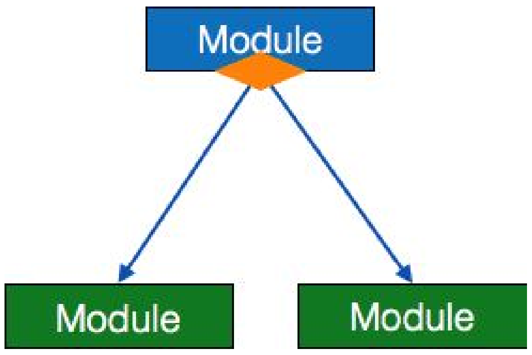
Here are the symbols used in construction of structure charts -

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



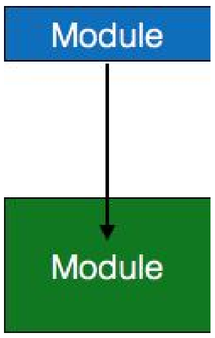
**Representation of Structured charts for Library System**

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

- 

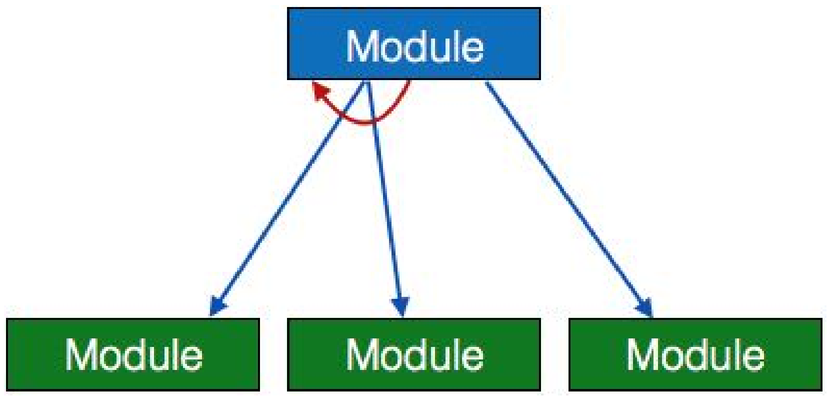
**Representation of Condition**

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



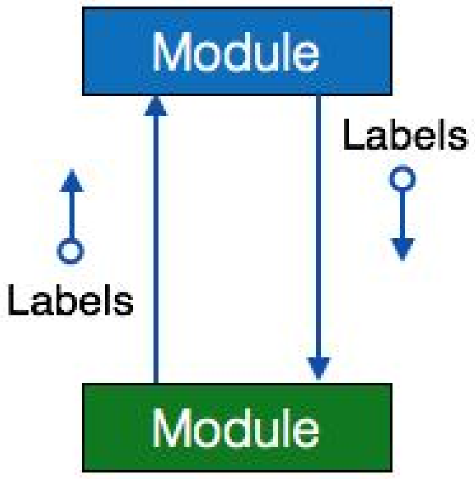
**Representation of jump**

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



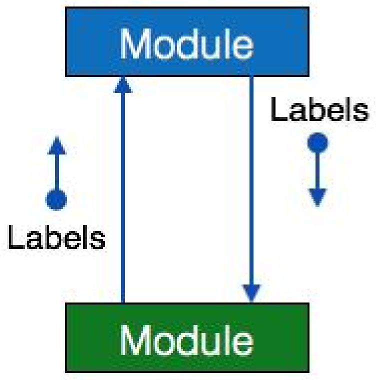
**Representation of loop**

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



**Representation of Data flow**

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



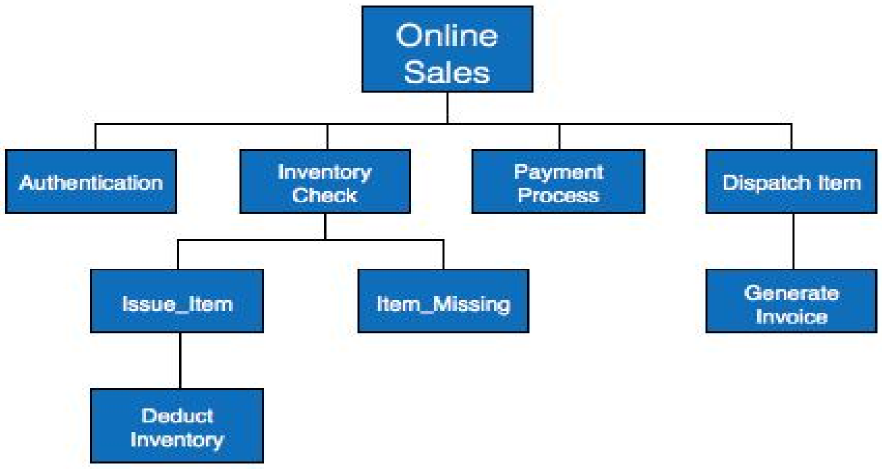
**Representation of control flow**

## 6.5.4 HIPO Diagram

HIPO (Hierarchical Input Process Output) diagram is a combination of two organized method to analyze the system and provide the means of documentation. HIPO model was developed by IBM in year 1970.

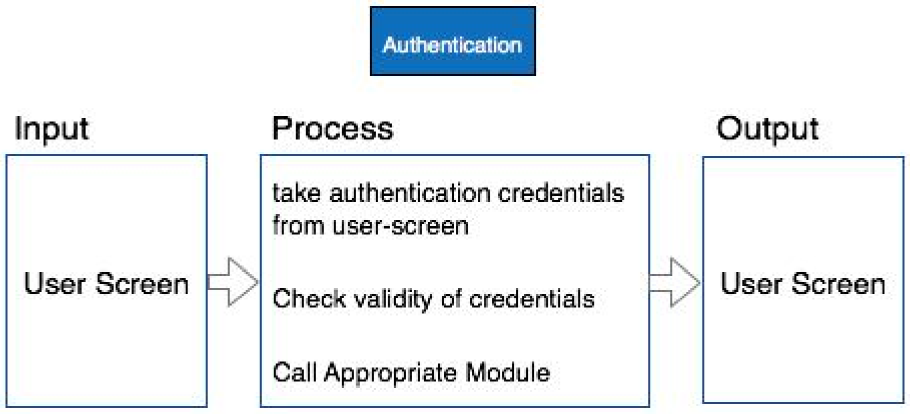
HIPO diagram represents the hierarchy of modules in the software system. Analyst uses HIPO diagram in order to obtain high-level view of system functions. It decomposes functions into sub-functions in a hierarchical manner. It depicts the functions performed by system.

HIPO diagrams are good for documentation purpose. Their graphical representation makes it easier for designers and managers to get the pictorial idea of the system structure.



**HPO Diagram for Online shopping**

In contrast to IPO (Input Process Output) diagram, which depicts the flow of control and data in a module, HIPO does not provide any information about data flow or control flow.


Authentication
Input
Process
Output
User Screen
take authentication credentials from user-screen
Check validity of credentials
Call Appropriate Module
User Screen

**Input Process Output diagram**

**Example**

Both parts of HIPO diagram, Hierarchical presentation and IPO Chart are used for structure design of software program as well as documentation of the same.

### 6.5.5 Structured English

Most programmers are unaware of the large picture of software so they only rely on what their managers tell them to do. It is the responsibility of higher software management to provide accurate information to the programmers to develop accurate yet fast code.

Other forms of methods, which use graphs or diagrams, may are sometimes interpreted differently by different people.

Hence, analysts and designers of the software come up with tools such as Structured English. It is nothing but the description of what is required to code and how to code it. Structured English helps the programmer to write error-free code.

Other form of methods, which use graphs or diagrams, may are sometimes interpreted differently by different people. Here, both Structured English and Pseudo-Code tries to mitigate that understanding gap.

Structured English is the It uses plain English words in structured programming paradigm. It is not the ultimate code but a kind of description what is required to code and how to code it. The following are some tokens of structured programming.

IF-THEN-ELSE,

DO-WHILE-UNTIL

Analyst uses the same variable and data name, which are stored in Data Dictionary, making it much simpler to write and understand the code.

**Example**

We take the same example of Customer Authentication in the online shopping environment. This procedure to authenticate customer can be written in Structured English as:

Enter Customer_Name

```
SEEK Customer_Name in Customer_Name_DB file
IF Customer_Name found THEN
   Call procedure USER_PASSWORD_AUTHENTICATE()
ELSE
   PRINT error message
   Call procedure NEW_CUSTOMER_REQUEST()
ENDIF
```

The code written in Structured English is more like day-to-day spoken English. It cannot be implemented directly as a code of software. Structured English is independent of programming language.

### 6.5.6 Pseudo-Code

Pseudo code is written more close to programming language. It may be considered as augmented programming language, full of comments and descriptions.

Pseudo code avoids variable declaration but they are written using some actual programming language's constructs, like C, Fortran, Pascal etc.

Pseudo code contains more programming details than Structured English. It provides a method to perform the task, as if a computer is executing the code.

**Example**

Program to print Fibonacci up to n numbers.

```
void function Fibonacci
Get value of n;
Set value of a to 1;
Set value of b to 1;
```

```
Initialize I to 0

for (i=0; i< n; i++)

{

  if a greater than b

  {

    Increase b by a;

    Print b;

  }

  else if b greater than a

  {

    increase a by b;

    print a;

  }

}
```

## 6.5.7 Decision Tables

A Decision table represents conditions and the respective actions to be taken to address them, in a structured tabular format.

It is a powerful tool to debug and prevent errors. It helps group similar information into a single table and then by combining tables it delivers easy and convenient decision-making.

**Creating Decision Table**

To create the decision table, the developer must follow basic four steps:

- Identify all possible conditions to be addressed
- Determine actions for all identified conditions

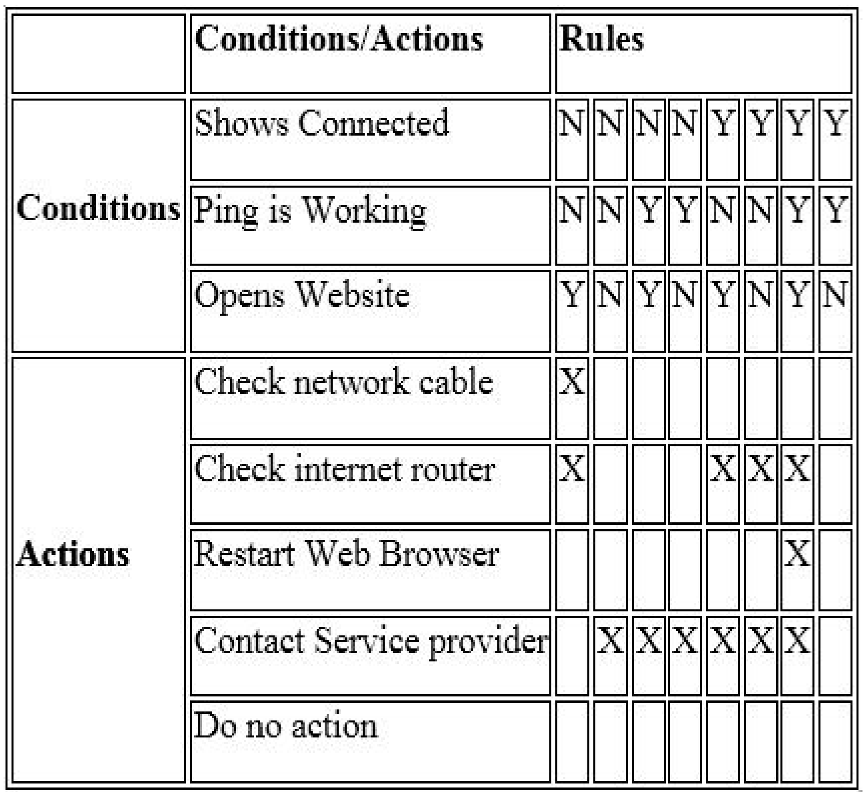
- Create Maximum possible rules
- Define action for each rule

Decision Tables should be verified by end-users and can lately be simplified by eliminating duplicate rules and actions.

**Example**

Let us take a simple example of day-to-day problem with our Internet connectivity. We begin by identifying all problems that can arise while starting the internet and their respective possible solutions.

We list all possible problems under column conditions and the prospective actions under column Actions.

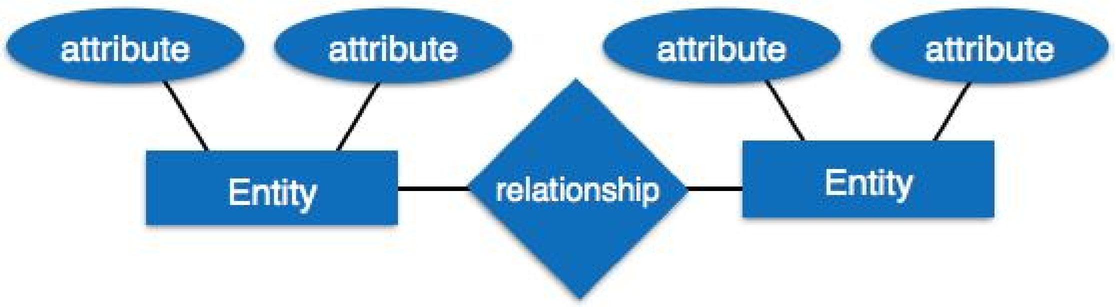


| | **Conditions/Actions** | **Rules** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Conditions** | Shows Connected | N | N | N | N | Y | Y | Y | Y |
| | Ping is Working | N | N | Y | Y | N | N | Y | Y |
| | Opens Website | Y | N | Y | N | Y | N | Y | N |
| **Actions** | Check network cable | X | | | | | | | |
| | Check internet router | X | | | | X | X | X | |
| | Restart Web Browser | | | | | | | X | |
| | Contact Service provider | | X | X | X | X | X | X | |
| | Do no action | | | | | | | | |

## 6.6 Entity-Relationship Model

Entity-Relationship(ER) model is a type of database model based on the notion of real world entities and relationship among them. We can map real world scenario onto ER database model. ER Model creates a set of entities with their attributes, a set of constraints and relation among them.

ER Model is best used for the conceptual design of database. ER Model can be represented as follows :



**ER Model**

- **Entity** - An entity in ER Model is a real world being, which has some properties called **attributes**. Every attribute is defined by its corresponding set of values, called **domain**.

  For example, Consider a school database. Here, a student is an entity. Student has various attributes like name, id, age and class etc.

- **Relationship** - The logical association among entities is called **relationship**. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of associations between two entities.

  Mapping cardinalities:

  - one to one
  - one to many
  - many to one
  - many to many

## 6.7 Data Dictionary

Data dictionary is the centralized collection of information about data. It stores meaning and origin of data, its relationship with other data, data format for usage etc. Data dictionary has rigorous definitions of all names in order to facilitate user and software designers.

Data dictionary is often referenced as meta-data (data about data) repository. It is created along with DFD (Data Flow Diagram) model of software program and is expected to be updated whenever DFD is changed or updated.

**Requirement of Data Dictionary**

The data is referenced via data dictionary while designing and implementing software. Data dictionary removes any chances of ambiguity. It helps keeping work of programmers and designers synchronized while using same object reference everywhere in the program.

Data dictionary provides a way of documentation for the complete database system in one place. Validation of DFD is carried out using data dictionary.

**Contents**

Data dictionary should contain information about the following

- Data Flow
- Data Structure
- Data Elements
- Data Stores
- Data Processing

Data Flow is described by means of DFDs as studied earlier and represented in algebraic form as described.

| | |
|---|---|
| **=** | **Composed of** |
| {} | Repetition |
| () | Optional |

+ And

[ / ] Or

Example

Address = House No + (Street / Area) + City + State

Course ID = Course Number + Course Name + Course Level + Course Grades

**Data Elements**

Data elements consist of Name and descriptions of Data and Control Items, Internal or External data stores etc. with the following details:

- Primary Name
- Secondary Name (Alias)
- Use-case (How and where to use)
- Content Description (Notation etc. )
- Supplementary Information (preset values, constraints etc.)

**Data Store**

It stores the information from where the data enters into the system and exists out of the system. The Data Store may include -

- **Files**
  - Internal to software.
  - External to software but on the same machine.
  - External to software and system, located on different machine.
- **Tables**
  - Naming convention
  - Indexing property

**Data Processing**

There are two types of Data Processing:

- **Logical:** As user sees it
- **Physical:** As software sees it

## 6.8 Behavioral Model

Behavioral modeling is an operational principle for all requirements analysis methods. The behavioral model indicates how software will respond to external events or stimuli.

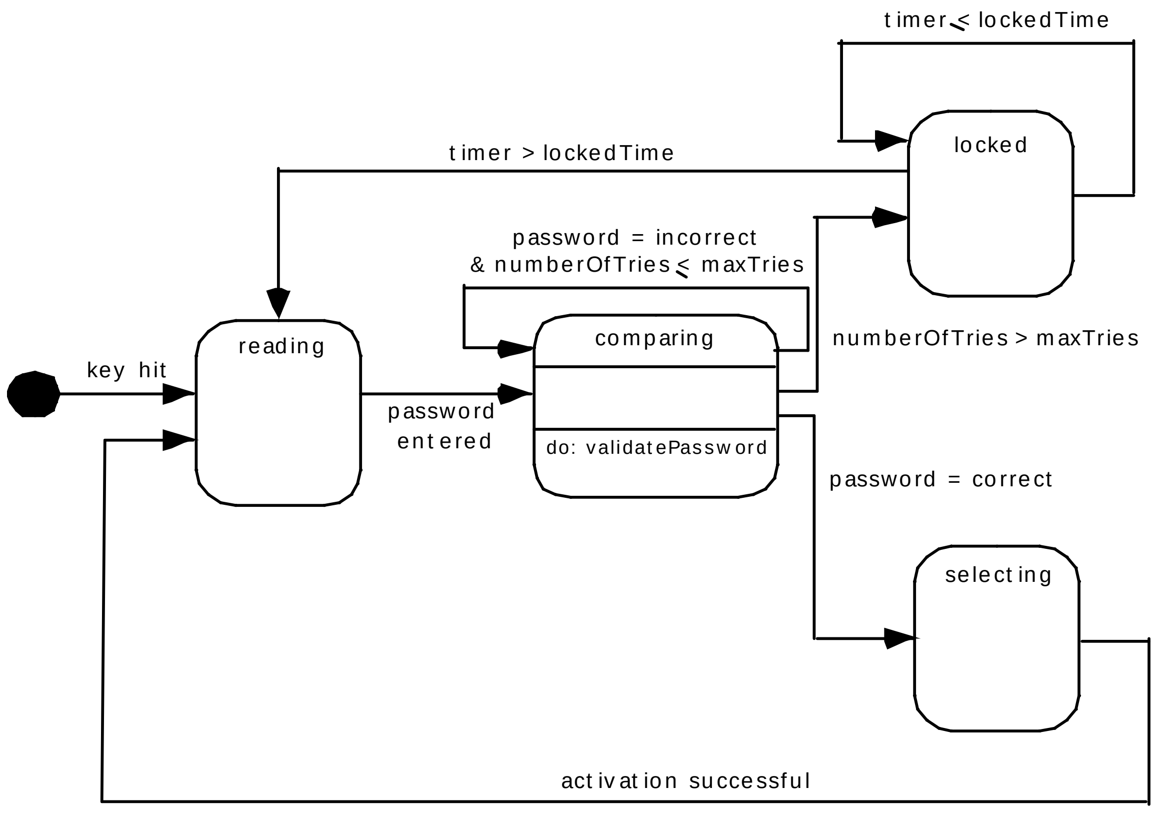
To create the model, the analyst must perform the following steps:

1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create a sequence for each use-case.
4. Build a state diagram for the system.
5. Review the behavioral model to verify accuracy and consistency.

**State Representations**

- In the context of behavioral modeling, two different characterizations of states must be considered:
    - the state of each class as the system performs its function and
    - the state of the system as observed from the outside as the system performs its function
- The state of a class takes on both passive and active characteristics.
    - A passive state is simply the current status of all of an object’s attributes.

The active state of an object indicates the current status of the object as it undergoes a continuing transformation or processing.



**State Diagram for Alarm system**

**The States of a System**

- State—a set of observable circumstances that characterizes the behavior of a system at a given time
- State transition—the movement from one state to another
- Event—an occurrence that causes the system to exhibit some predictable form of behavior
- Action—process that occurs as a consequence of making a transition

## 6.8.1 Behavioural Modeling in System Engineering

The System engineering is understood as complex discipline for the system design and analysis of the system. The system is for the purpose of the system engineering defined as a set of the components which are interconnected and provides the group of emergent properties . These emergent properties are derivate from the properties of the system components, but new dimension is added by the system integration.

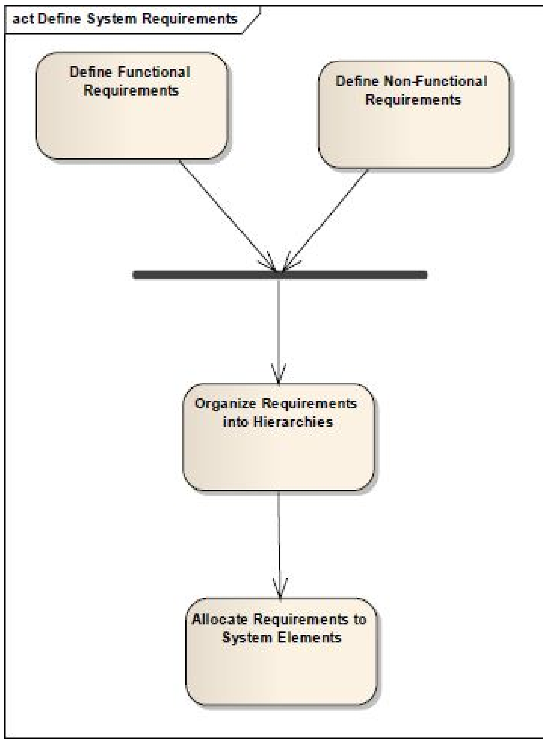
A system engineering processes are about preparing generic stakeholder goals, requirements, system design, evaluation of alternative system designs, allocation of functional requirements, system verification. All of these activities lead to creation of the balanced system.

The system development generic process is described by waterfall model . The process should be composed of:

1) Requirements Engineering
2) Definition and Elicitation.
3) System Design.
4) Sub-system Development.
5) System Integration.
6) System Installation.
7) System Evolution.
8) System Decommissioning.

The system requirements engineering general name for the specific sets of the software engineering techniques, which is used at the beginning of the software cycle. The purpose of these techniques is to discover stakeholder’s needs.

The requirements gathering process in system engineering has these basics steps, which are graphically described in the figure below.


act Define System Requirements
Define Functional Requirements
Define Non-Functional Requirements
Organize Requirements into Hierarchies
Allocate Requirements to System Elements

**Requirements Definition Model**

The requirement  is a description of the functionality or condition which stakeholders define for the system. After the first round of requirements gathering is talked about the raw or abstract requirements. These raw requirements are list of functionality or condition for the proposed software, which is unanalyzed yet. The most important in this phase is to establish the project goals, which should be achieved.

The next group of the requirements is non- functional requirements. The purpose of these requirements is familiarized system designers with problem domain and conditions in the domain. Well-known examples of these are reliability, performance, safety or security. These non- functional requirements are critical in the system evaluation very often.

The last part of the requirements gathering is so- called system characteristics. The system characteristics are commonly prepared in negative way. It means, that system design specifies what irrelevant system behavior is.

The system design phase deals with association of the system requirements to individual sub- system, more specific to system components. The set of the system requirements is studied and individual requirement is associated to proposed component. One of the most common techniques, which could be applied in this process, is to sub- system design first. Than system designers is able to associated selected requirements to the specific subsystem. The process of the sub-systems identification and requirements association is interactive and is commonly modeled in form of spiral.

In the system modeling phase is a system architecture designed. This activity is based on sub- system (or component) design. The system is modeled as group of interconnected blocks, where connection indicates data flow or other form of dependency. The main task is to create more concrete definition of the sub-systems, which were set-up in the system design phase.

The sub-system development phase works with sub-system or system components implementation. The sub-systems are implemented in parallel. This is because of sub-systems in the system engineering is not only hardware/software, but also should be create by civil engineering.

Next step in the system engineering life-cycle is the system integration. The implemented subsystem are integrated into the system . The integration is an incremental process. The sub-system are integrated, when their implementation is finished. This approach is time-less consuming, then legacy approach, when all components were implemented together.

The system evolution and system decommission are the final phase of the system lifecycle. A system designer should care about system improvements during its production time. They also should take care about times, when system should be prepared for out-of-service elaboration.

The main task of the software engineering is a system development from very beginning (requirements elicitation) to system decommission.

## 1 Problem Formulation

In the system engineering lifecycle, which were described above, basic ideas of the system engineering were concluded. In the system engineering are used well-known techniques for modeling. Probably the most common is block diagramming. This model is used for system design and for sub-system analyze. Secondly data flow diagrams are common for definition of data and process designing.

Today state-of-art in research of graphical documentation of the system development is System Modeling Language (SysML). The SysML is a graphical modeling language , which is derivate from Unified Modeling Language (UML). UML is an industry standard in the scope of the software engineering.

SysML is a extension of UML, this two basic technique shared basic principles and some types of diagrams are used in both. The SysML take important role in the system engineering, because its usability in all phase of software engineering process.

### 2 System Modeling Language

SysML is relatively young modeling language. Its history is written from 2001, when Systems Engineering Domain Special Interest Group were set-up [3]. Today, there is 1.2 version from 2010 valid.

## 6.9 Language Overview

The modeling project support analysis, specification, design, verification and validation of systems. SysML therefore support all phases of the system engineering lifecycle. The system components should be described by structural composition, interconnection and classification.

Secondly by function-based approach which is based on messages between objects. This aspect should respect to constrains, which are derived from physical system structure or from performance properties. Important role takes association of system functionality to each of the system components.

In the SysML nine of diagrams are of recognized .

Requirements diagram is used for graphical interpretation of requirements and their connection to other requirements and to other elements and entities in project – such test cases, use cases. For those how are familiars to UML, this diagram is new in SysML and have never be used in UML.

Activity diagram is based on UML activity diagram; there is slightly different usage of it. Basically is used for modeling of flows of actions based on the availability of inputs, outputs or control. Transformation of actions is also modeled by activity diagram in SysML.

Sequence diagram is used for representing of message flow between objects. State machine diagram presents set of state of the modeled entity and events which generated message upon which entity state is switched.

Use Case diagram is used for the system function description. This model is composed of the actors, which are external entity and of the use cases. The use cases represent system functions or algorithms. Each of the use case has to realize one of the requirements as minimum.

Block definition diagram is used instead of the class model in UML. The purpose of the block definition is to model system structure.

Internal block diagram is similar to UML composite structure. The main idea of this diagram is to model internal structure of the each individual part of the proposed block. Very important here is modeling of interface and communication between block´s entities.

Parametric diagram is SysML origin and have no predecessor in UML principles. Is used for modeling system parameters and constrains, should be used for critical – hazard system states.

Package diagram is useful for model organizing. Model elements should clustered by its stereotypes. Packages also should be used for creation of a large project structure.

## Check Your Progress

1. **Fill in the blanks** DFD also called as __________
2. Data flow diagram is ____________representation of flow of data in an information system.
3. **Choose the correct answer** Structure chart represents hierarchical structure of __________

    a. Data b. information c. modules d. code

4. HIPO diagrams are good for ___________purpose.

    a. Coding b. design c. testing d. documentation

5. **Say True or False** Pseudo code contains more programming details than Structured English.

## 6.10 Behavioral Modeling Process

In this lesson will be introduced sample projects, which contains example model of each type of the diagrams. For purpose of this article authors adopted project which was prepared by SparxSystems as sample project in SysML language .

In this sample is described development of the audio Player. The main task of the project team was to offer a solution which was successfully in usability and which offers appropriate functionality. For purpose of this description authors used modified version of behavioral modeling process . The behavioral modeling is a description (Figure 3 how the proposed system will interact with the actors and with entitles which is out of boundary of the system.

The first step in the creation of the system behavioral model is the requirements gathering. In the figure can be seen model of the requirements. The diagram illustrates hierarchical structure of the requirements. On the top of the tree the Specification package can be seen. Other parts are interconnected by containment association. The specification is clustered into sever groups. Groups are User Friendliness, Durability, Performance and Media Capacity

The user friendliness group defines set of the requirements, which deal with quality of service of the audio player. Keys Layout, Graphical User Interface and Scroller are primary requirements, which take important role in user satisfaction.

When the requirements model is finished, next step is use case model. In the use case model there is an algorithmic definition of the actors' activity in the system. In the figure system boundary, actor and use cases can be seen. The boundary is named as Playlist Maintenance, because this model describes only activities of the listener, which are available as maintain the playlist. Inside of the each individual use case is scenario description.

The scenario is a sequence of steps, which describes a track, should be downloaded. In this Use Case model include association is used. The “include” association is used for situation, when the use case contains other use case for achieving its goal.

The Use Case models are derivate from requirements. The Use Cases are commonly prepared for individual system blocks. This is rigorous connection between the requirement and the use case. This connection should be modeled as realization in the use case model or can be documented in form of the responsibility matrix.

The interaction in form of sequence diagram is useful for modeling overview of operations. It describes all possible activity of the actor, called Listener. There are used fragments, which used for referencing individual activity descriptions. These are named as ref. The alt abbreviation is used for conditions. In this context the view new tracks can be executed only if the playlist exists.

The last view of the system can be done by state machine diagram. The most important noticeable fact here is, that by state machine diagram is modeled same system in the different view only. The modified version of the state machine diagram. It does not present only two basic states – idle and connected. The state connected is describes in form of the activity diagram. In the connected state tracks could be copied and downloaded. These two operations are in a parallel section, which is created by using fork/join artifacts.

## 6.11 Summary

The guidelines for drawing DFD during the design phase have been read in this lesson. The representation of symbols used in different levels of DFD is also clearly described . The ER model and behavioural model and its procedure are discussed in detail.

## 6.12 Check your Answer

**1.** context level diagram

**2.** graphical

**3.** modules.

4. Documentation

5. True

## 6.13 Model Questions

1. What is the purpose of using DFD?
2. Write short notes on Data Dictionary.
3. What are the steps involved to draw the DFD diagram ?
4. Compare and contrast between ER model and Behavioral model.
5. Write briefly about Behavioral Modeling Process with example.

# LESSON-7

# THE DESIGN MODEL

## 7.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- Understand the design model and its steps involved in it.
- Understand the terminologies like Data Modeling, Data Structures, Databases and the Data Warehouse .
- Use transform mapping during the design phase .

### Structure

## 7.2 Introduction

In this lesson, Design model has been discussed in detail. It discuss about the types of design models like deployment model and component model . It further discuss about the architecture styles used in the design phase .

## 7.3 The Design Model

A **design model** in Software Engineering is an object-based picture or pictures that represent the use cases for a system. Or to put it another way, it is the means to describe a system’s implementation and source code in a diagrammatic fashion. This type of representation has a couple of advantages. First, it is a simpler representation than words alone. Second, a group of people can look at these simple diagrams and quickly get the general idea behind a system. In the end, it boils down to the old adage, ‘a picture is worth a thousand words.

### 7.3.1 Types of Design Model

**1. Data design elements**

- The data design element produced a model of data that represent a high level of abstraction.
- This model is then more refined into more implementation specific representation which is processed by the computer based system.
- The structure of data is the most important part of the software design.

**2. Architectural design elements**

- The architecture design elements provides us overall view of the system.
- The architectural design element is generally represented as a set of interconnected subsystem that are derived from analysis packages in the requirement model.

**The architecture model is derived from following sources:**

- The information about the application domain to build the software.
- Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.

- The architectural style and pattern as per availability.

**3. Interface design elements**

- The interface design elements for software represents the information flow within it and out of the system.
- They communicate between the components defined as part of architecture.

**Following are the important elements of the interface design:**

1. The user interface

2. The external interface to the other systems, networks etc.

3. The internal interface between various components.

**4. Component level diagram elements**

- The component level design for software is similar to the set of detailed specification of each room in a house.
- The component level design for the software completely describes the internal details of the each software component.
- The processing of data structure occurs in a component and an interface which allows all the component operations.
- In a context of object-oriented software engineering, a component shown in a UML diagram.
- The UML diagram is used to represent the processing logic.

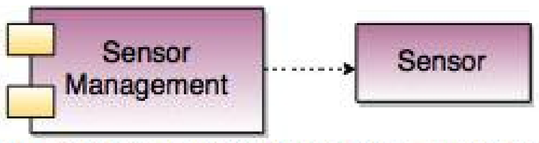
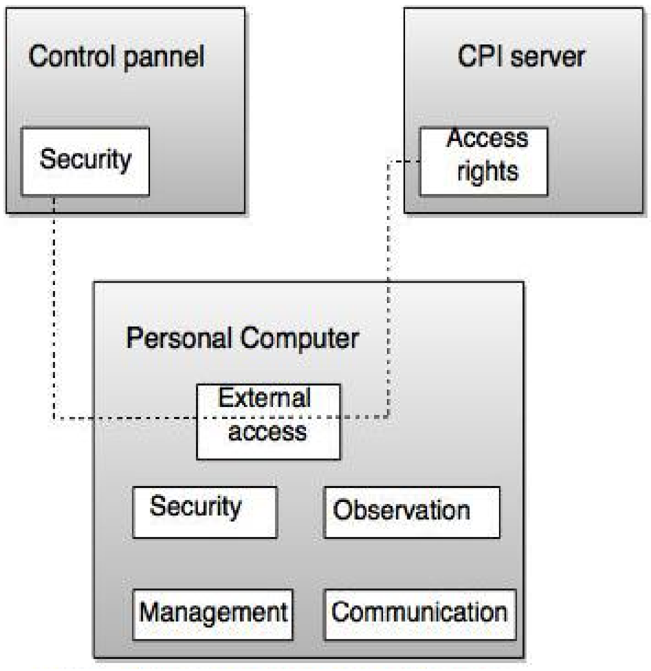


**Fig. - UML component diagram for sensor managemnet**

**UML Component diagram for sensor management**

**5. Deployment level design elements**

- The deployment level design element shows the software functionality and subsystem that allocated in the physical computing environment which support the software.
- Following figure shows three computing environment as shown. These are the personal computer, the CPI server and the Control panel.



**ARCHITECTURAL DESIGN**

## 7.3.2 What Is Architecture?

When we discuss the architecture of a building, many different attributes come to mind. At the most simplistic level, we consider the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colours, and materials are combined to create the external facade and the internal “living environment.” It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art. But what about software architecture?

Bass, Clements, and Kazman [BAS98] define this elusive term in the following way. The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to (1) analyse the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reducing the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified.

The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol. The design of software architecture considers two levels of the design : pyramid data design and architectural design.

In the context of the preceding discussion, data design enables us to represent the data component of the architecture. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

### 7.3.3 Why Is Architecture Important?

Bass and his colleagues {BAS98] identify three key reasons that software architecture is important:

• Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer- based system.

• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

• Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"[BAS98].The architectural design model and the architectural patterns contained within it are transferrable. That is, architecture styles and patterns can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

## 7.4 Data Design

Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

### 7.4.1 Data Modeling, Data Structures, Databases and the Data Warehouse

The data objects defined during software requirements analysis are modelled using entity/ relationship diagrams and the data dictionary .The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level. In years past, data architecture was generally limited to data structures at the program level and databases at the application level. But today, businesses large and small are awash in data. It is not unusual for even a moderately sized business to have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in databases (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment. An alternative solution, called a data warehouse, adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business. In a sense, a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business. But many characteristics differentiate a data warehouse from the typical database.

**Subject orientation**

A data warehouse is organized by major business subjects, rather than by business process or function. This leads to the exclusion of data that may be necessary for a particular business function but is generally not necessary for data mining.

**Integration**

Regardless of the source, the data exhibit consistent naming conventions, units and measures, encoding structures, and physical attributes, even when inconsistency exists across different application-oriented databases.

**Time Variant**

For a transaction-oriented application environment, data are accurate at the moment of access and for a relatively short time span (typically 60 to 90 days) before access. For a data warehouse, however, data can be accessed at a specific moment in time (e.g., customers contacted on the date that a new product was announced to the trade press). The typical time horizon for a data warehouse is five to ten years.

**Non-volatility**

Unlike typical business application databases that undergo a continuing stream of changes (inserts, deletes, updates), data are loaded into the warehouse, but after the original transfer, the data do not change.

### 7.4.2 Data Design at the Component Level

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. Wasserman [WAS80] has proposed a set of principles that may be used to specify and design such data structures. In actuality, the design of data begins during the creation of the analysis model. Recalling that requirements analysis and design often overlap, we consider the following set of principles [WAS80] for data specification:

1. **The systematic analysis principles applied to function and behaviour should also be applied to data.**

We spend much time and effort deriving, reviewing, and specifying functional requirements and preliminary design. Representations of data flow and content should also be developed and reviewed, data objects should be identified, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated.

For example, specification of a multipronged linked list may nicely satisfy data requirements but lead to an unwieldy software design. An alternative data organization may lead to better results.

**2. All data structures and the operations to be performed on each should be identified.**

The design of an efficient data structure must take the operations to be performed on the data structure into account (e.g., see [AHO83]). For example, consider a data structure made up of a set of diverse data elements. The data structure is to be manipulated in a number of major software functions. Upon evaluation of the operations performed on the data structure, an abstract data type is defined for use in subsequent software design. Specification of the abstract data type may simplify software design considerably.

**3. A data dictionary should be established and used to define both data and program design.**

A data dictionary explicitly represents the relationships among data objects and the constraints on the elements of a data structure. Algorithms that must take advantage of specific relationships can be more easily defined if a dictionarylike data specification exists.

**4.Low-level data design decisions should be deferred until late in the design process**.

A process of stepwise refinement may be used for the design of data. That is, overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component level design. The top-down approach to data design provides benefits that are analogous to a top-down approach to software design—major structural attributes are designed and evaluated first so that the architecture of the data may be established.

**5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.**

The concept of information hiding and the related concept of coupling provide important insight into the quality of a software design. This principle alludes to the importance of these concepts as well as "the importance of separating the logical view of a data object from its physical view" [WAS80].

**6. A library of useful data structures and the operations that may be applied to them should be developed**.

Data structures and operations should be viewed as a resource for software design. Data structures can be designed for reusability. A library of data structure templates (abstract data types) can reduce both specification and design effort for data.

**7. A software design and programming language should support the specification and realization of abstract data types.**

The implementation of a sophisticated data structure can be made exceedingly difficult if no means for direct specification of the structure exists in the programming language chosen for implementation. These principles form a basis for a component-level data design approach that can be integrated into both the analysis and design activities.

## 7.5 Architectural Styles

When a builder uses the phrase "centre hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an architectural style as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a pattern for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the pattern—a “centre hall colonial”— guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

(1) a set of components (e.g., a database, computational modules) that perform a function required by a system;

(2) a set of connectors that enable “communication, coordination’s and cooperation” among components;

(3) constraints that define how components can be integrated to form the system;

(4) semantic models that enable a designer to understand the overall properties of a system by analysing the known properties of its constituent parts [BAS98]. In the section that follows, we consider commonly used architectural patterns for software.

### 7.5.1 A Brief Taxonomy of Styles and Patterns

Although millions of computer-based systems have been created over the past 50 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

**Data-centred architectures**

A data store (e.g., a file or database) resides at the centre of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.

Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client change.

**Data-flow architectures**

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe and filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.

Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighbouring filters.

If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.

**Call and return architectures**

This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within this category:

**Main program/subprogram architectures**

This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components.

**Object-oriented architectures**

The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

**Layered architectures**

A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available to the software designer. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

## 7.5.2 Organization and Refinement

Because the design process often leaves a software engineer with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [BAS98] provide insight into the architectural style that has been derived:

**Control**

How is control managed within the architecture?

Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?

How do components transfer control within the system?

How is control shared among components?

What is the control topology (i.e., the geometric form3 that the control takes)?

Is control synchronized or do components operate asynchronously?

**Data**

How are data communicated between components?

Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)?

Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?

How do functional components interact with data components?

Are data components passive or active (i.e., does the data component actively interact with other components in the system)?

How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more-detailed analysis of the architecture.

## 7.6 Analyzing Alternative Architectural Designs

The questions posed in the preceding section provide a preliminary assessment of the architectural style chosen for a given system. However, a more complete method for evaluating the quality of an architecture is essential if design is to be accomplished effectively. In the sections that follow, we consider two different approaches for the analysis of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

### 7.6.1 An Architecture Trade-off Analysis Method

The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) [KAZ98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

**Collect scenario**

A set of use-cases is developed to represent the system from the user’s point of view.

**Elicit requirements, constraints, and environment description.**

This information is required as part of requirements engineering and is used to be certain that all customer, user, and stakeholder concerns have been addressed.

**Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.**

The style(s) should be described using architectural views such as

• Module view for analysis of work assignments with components and the degree to which information hiding has been achieved.

• Process view for analysis of system performance.

• Data flow view for analysis of the degree to which the architecture meets functional requirements.

**Evaluate quality attributes by considering each attribute in isolation**

The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.

**Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style**

This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.

**Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5**.

The SEI describes this approach in the following manner [KAZ98]:Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive.

For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The availability of that architecture might also vary directly with the number of servers. However, the security of the system might vary inversely with the number of servers (because the system contains more potential points of attack). The number of servers, then, is a trade-off point with respect to this architecture.

It is an element, potentially one of many, where architectural trade-offs will be made, consciously or unconsciously.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.

## 7.6.2 Quantitative Guidance for Architectural Design

One of the many problems faced by software engineers during the design process is a general lack of quantitative methods for assessing the quality of proposed designs. Work in the area of quantitative analysis of architectural design is still in its formative stages. Asada and his colleagues [ASA96] suggest a number of pseudo quantitative techniques that can be used to complement the ATAM approach as a method for the analysis of architectural design quality. Asada proposes a number of simple models that assist a designer in determining the degree to which a particular architecture meets predefined “goodness” criteria.

These criteria, sometimes called design dimensions, often encompass the quality attributes defined in the last section: reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability, among others. The first model, called spectrum analysis, assesses an architectural design on a “goodness” spectrum from the best to worst possible designs. Once the software architecture has been proposed, it is assessed by assigning a “score” to each of its design dimensions. These dimension scores are summed to determine the total score, S, of the design as a whole. Worst-case scores4 are then assigned to a hypothetical design, and a total score, Sw, for the worst case architecture is computed.

A best-case score, Sb, is computed for an optimal design. We then calculate a spectrum index, Is, using the equation

Is = [(S _ Sw)/(Sb _ Sw)] _ 100

The spectrum index indicates the degree to which a proposed architecture approaches an optimal system within the spectrum of reasonable choices for a design. If modifications are made to the proposed design or if an entirely new design is proposed, the spectrum indices for both may be compared and an improvement index, Imp, may be computed:

Imp = Is1 _ Is2

This provides a designer with a relative indication of the improvement associated with architectural changes or a new proposed architecture. If Imp is positive, then we can conclude that system 1 has been improved relative to system 2.

**Design selection analysis** is another model that requires a set of design dimensions to be defined. The proposed architecture is then assessed to determine the number of design dimensions that it achieves when compared to an ideal (best-case) system.

For example, if a proposed architecture would achieve excellent component reuse, and this dimension is required for an idea system, the reusability dimension has been achieved. If the proposed architecture has weak security and strong security is required, that design dimension has not been achieved.

We calculate a design selection index, d, as

d = (Ns/Na) _ 100

where Ns is the number of design dimensions achieved by a proposed architecture and Na is the total number of dimensions in the design space. The higher the design selection index, the more closely the proposed architecture approaches an ideal system.

**Contribution analysis** "identifies the reasons that one set of design choices gets a lower score than another" [ASA96]. A set of "realization mechanisms" (features of the architecture) are identified. All customer requirements (determined using QFD) are listed and a cross-reference matrix is created. The cells of the matrix indicate the relative strength of the relationship (on a numeric scale of 1 to 10) between a realization mechanism and a requirement for each alternative architecture. This is sometimes called a quantified design space (QDS).

The QDS is relatively easy to implement as a spreadsheet model and can be used to isolate why one set of design choices gets a lower score than another.

## 7.6.3 Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao [ZHA98] suggests three types of dependencies:

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.

Flow dependencies represent dependence relationships between producers and consumers of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.

Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.

## 7.7 Mapping Requirements INTO a Software rchitecture

Structured design is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process:

(1) the type of information flow is established;

(2) flow boundaries are indicated;

(3) the DFD is mapped into program structure;

(4) control hierarchy is defined;

(5) resultant structure is refined using design measures and heuristics;

(6) the architectural description is refined and elaborated.

The type of information flow is the driver for the mapping approach required in step 3. In the following sections we examine two flow types.

### 7.7.1 Transform Flow

Recalling the fundamental system model (level 0 data flow diagram), information must enter and exit software in an “external world” form. For example, data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information. Such externalized data must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as incoming flow. At the kernel of the software, a transition occurs. Incoming data are passed through a transform centre and begin to move along paths that now lead “out” of the software.

Data moving along these paths are called outgoing flow. The overall flow of data occurs in a sequential manner and follows one, or only a few, “straight line” paths. When a segment of a data flow diagram exhibits these characteristics, transform flow is present. It is also important to note that the call and return architecture can reside within other more sophisticated architectures discussed earlier in this chapter. For example, the architecture of one or more components of a client/server architecture might be call and return.

It should be noted that other elements of the analysis model (e.g., the data dictionary, PSPECs, CSPECs) are also used during the mapping method. There are many cases, however, where the data flow architecture may not be the best choice for a complex system. Examples include systems that will undergo substantial change over time or systems in which the processing associated with the data flow is not necessarily sequential.

### 7.7.2 Transaction Flow

The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category. However, information flow is often characterized by a single data item, called a transaction, that triggers other data flow along one of many paths. Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and, based on its value, flow along one of many action paths is initiated. The hub of information flow from which many action paths emanate is called a transaction centre.

It should be noted that, within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

## 7.8 Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

### 7.8.1 Design Steps

The preceding example will be used to illustrate each step in transform mapping. The steps begin with a re-evaluation of work done during requirements analysis and then move to the design of the software architecture.

**Step 1. Review the fundamental system model**

The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the System Specification and the Software Requirements Specification. Both documents describe information flow and structure at the software interface.

**Step 2. Review and refine data flow diagrams for the software:**

Information obtained from analysis models contained in the Software Requirements Specification is refined to produce greater detail.

**Step 3. Determine whether the DFD has transform or transaction flow characteristics:**

In general, information flow within a system can always be represented as transform. In this step, the designer selects global (software wide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These Sub flows can be used to refine program architecture derived from a global characteristic described previously.

**Step 4. Isolate the transform centre by specifying incoming and outgoing flow boundaries:**

In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

**Step 5. Perform "first-level factoring."**

Program structure represents a top-down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing.

A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:

• An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.

• A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

• An outgoing information processing controller, called alarm output controller,

coordinates production of output information.

**Step 6. Perform "second-level factoring."**

Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure.

**Step 7. Refine the first-iteration architecture using design heuristics for improved software quality**

A first-iteration architecture can always be refined by applying concepts of module independence . Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion and maintained without grief.

## Check your Progress

1. **Fill in the blanks**The ___________design elements provides us overall view of the system

2. A data warehouse is organized by major business ________, rather than by business process or function.

3. **Choose the correct answer** ______________ "identifies the reasons that one set of design choices gets a lower score than another" .

   a. Design analysis b. architectural technique c. analysis method d. contribution analysis

4. The hub of information flow from which many action paths emanate is called ______.

   a. Internet b. flow control c. multi transaction centre d. transaction centre

5. **Say True or False** Data flow architecture may not be the best choice for a complex system.

## 7.9 Refining the Architectural Design

Successful application of transform or transaction mapping is supplemented by additional documentation that is required as part of architectural design. After the program structure has been developed and refined, the following tasks must be completed:

• A processing narrative must be developed for each module.

• An interface description is provided for each module.

• Local and global data structures are defined.

• All design restrictions and limitations are noted.

• A set of design reviews are conducted.

• Refinement is considered (if required and justified).

A processing narrative is (ideally) an unambiguous, bounded description of processing that occurs within a module. The narrative describes processing tasks, decisions, and I/O. The interface description describes the design of internal module interfaces, external system interfaces, and the human/computer interface. The design of data structures can have a profound impact on architecture and the procedural details for each software component. Restrictions and/or limitations for each module are also documented. Typical topics for discussion include restriction on data type or format, memory or timing limitations; bounding values or quantities of data structures; special cases not considered; specific characteristics of an individual module.

The purpose of a restrictions and limitations section is to reduce the number of errors introduced because of assumed functional characteristics. Once design documentation has been developed for all modules, one or more design reviews is conducted. The review emphasizes traceability to software requirements, quality of the software architecture, interface descriptions, data structure descriptions, implementation and test practicality, and maintainability.

## 7.10 Summary

The various design model  have been read in this lesson. The concept about the data design  is also clearly described . The various types of architecture styles are learnt with the steps involved .  Further you will know how to refine the architectural design.

## 7.11 Check Your Answers

1. architecture
2. subjects
3. Contribution analysis
4. transaction centre
5. True

## 7.12 Model Quesitons

1. List the elements involved in Component level diagram .

2. Write short notes on Taxonomy of Styles and Patterns.

3.Write briefly about transformation flow.

4. What are the steps involved in refining the architectural design ?

5. Write short notes on An Architecture Trade-off Analysis Method.

# LESSON-8

# COMPONENT LEVEL DESIGN AND USER INTERFACE

## 8.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- Design the component level diagram during the software development.
- Understand the different design notations used in the design phase.
- Learn to do design user interface .

## Structure

## 8.2 Introduction

In this lesson, explains about the component level design during the software development. It explains about the different types of notation viz., graphical design notation and tabular design notation. It further discuss about the user interface used during design phase.

## 8.3 Component Level Design

Component-level design, also called *procedural design,* occurs after data, architectural, and interface designs have been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low.

**Component-level design**, also known as **component-based software engineering (CBSE)** or **Component-Based Design (CBD)**, is an approach to software development that emphasizes the concept of reusable components. In CBSE, the preferred method is to create a system using 'off-the-shelf' components and employing a 'buy, don't build' mentality. Only when necessary are custom components built, as opposed to building the entire system from scratch.

The main rationale behind this approach is to significantly minimize system cost and deployment time. Other benefits include higher software quality (although sometimes this is debatable), and the ability to more easily identify software issues. Whether or not CBSE results in higher quality software is a topic for discussion because the quality of a single component does not necessarily reflect the quality of the system as a whole. On the other hand, issues are more easily detected because they can be isolated to a specific component. In the case of a custom built component, this is harder to do.

**Steps for Component-Level Design**

The development of any software system or application follows roughly the same steps:

1. Requirements analysis and specification
2. System and software design
3. Implementation and unit testing

4. System integration
5. System verification and validation
6. Operation support and maintenance
7. Disposal (though this step is often omitted)

Each of these will be discussed in more detail the sections below.

### 8.3.1 Requirements Analysis and Specification

**Requirements analysis** in software engineering essentially identifies a problem and then determines the requirements the solution must meet to be considered successful. In CBSE, a main consideration at this stage is whether solution can be created or not using existing software components.

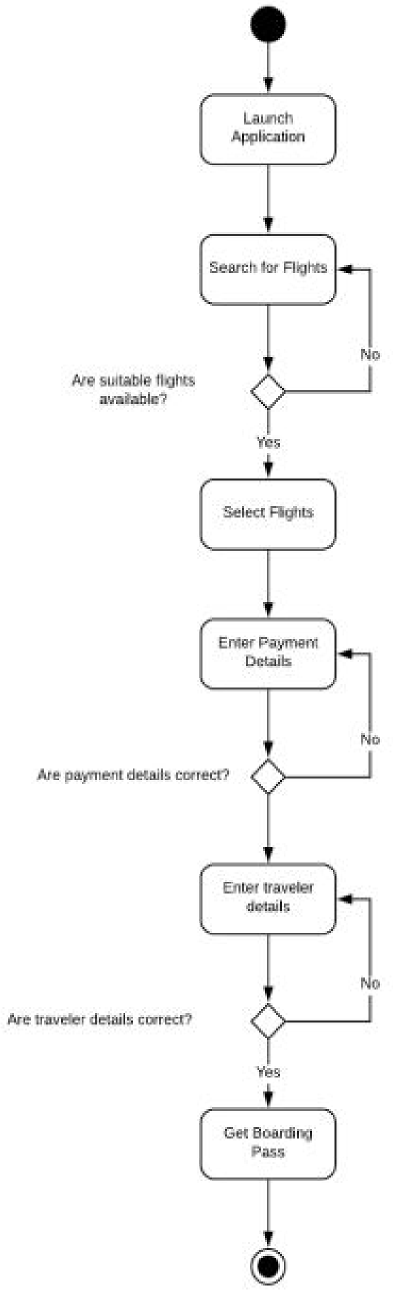
### 8.3.2 Requirements Modeling & Diagrams

Part of requirements analysis is **requirements modeling**, a multi-step process that produces diagrams and other graphical requirements representations. Most of these diagrams are drawn using **Unified Modeling Language (UML)**.

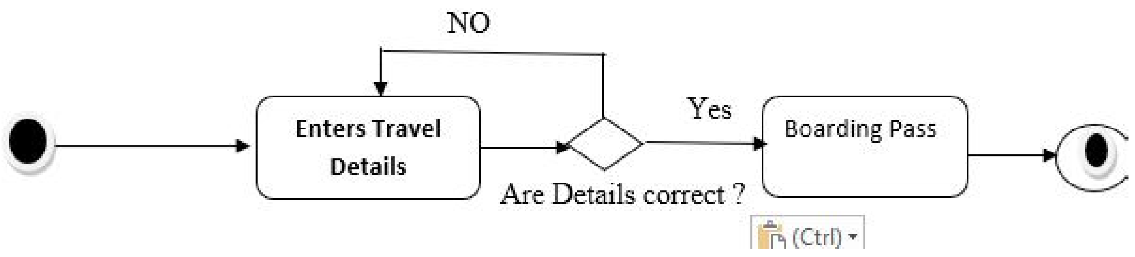
Two of the many diagrams produced during this stage are the **activity diagram** and the **state diagram**, often called state charts. An activity diagram is essentially an advanced flow chart, demonstrating how one activity in the system leads to another. A state diagram demonstrates how the system responds to external stimuli, like a user making choices through the application's interface.

Figures 1 and 2 show examples of activity and state diagrams for a proposed airline application that will allow the customer/traveler to search for flights, book tickets and print boarding passes.

The activity diagram above demonstrates the sequence or flow of how one possible activity leads to another using the example airline application.



Activity diagram for the proposed application



**State Diagram for the proposed application**

The state diagram demonstrates how the example airline application will respond for one specific use: the traveler getting a boarding pass.

### 8.3.3 System and Software Design

As in other approaches, **system design** establishes an overall system architecture. In CBSE, however, this is constrained by the availability of components. Theoretically, components from different models can be used, but in practice, all components should be from the same model (e.g. client-server) for best results.

### 8.3.4 Implementation and Unit Testing

In **implementation and unit testing**, the design is formalized and the individual units are tested. In CBSE, it is important to test not only individual components, but the assembly of components, as the components themselves might function correctly, but there may issues with the assembly. The component-level design can be represented by using different approaches. One approach is to use a programming language while other is to use some intermediate design notation such as graphical (DFD, flowchart, or structure chart), tabular (decision table), or text-based (program design language) whichever is easier to be translated into source code.

The component-level design provides a way to determine whether the defined algorithms, data structures, and interfaces will work properly. Note that a component (also known as **module)** can be defined as a modular building block for the software. However, the meaning of component differs according to how software engineers use it. The modular design of the software should exhibit the following sets of properties.

1. **Provide simple interface:** Simple interfaces decrease the number of interactions. Note that the number of interactions is taken into account while determining whether the software performs the desired function. Simple interfaces also provide support for reusability of components which reduces the cost to a greater extent. It not only decreases the time involved in design, coding, and testing but the overall software development cost is also liquidated gradually with several projects. A number of studies so far have proven that the

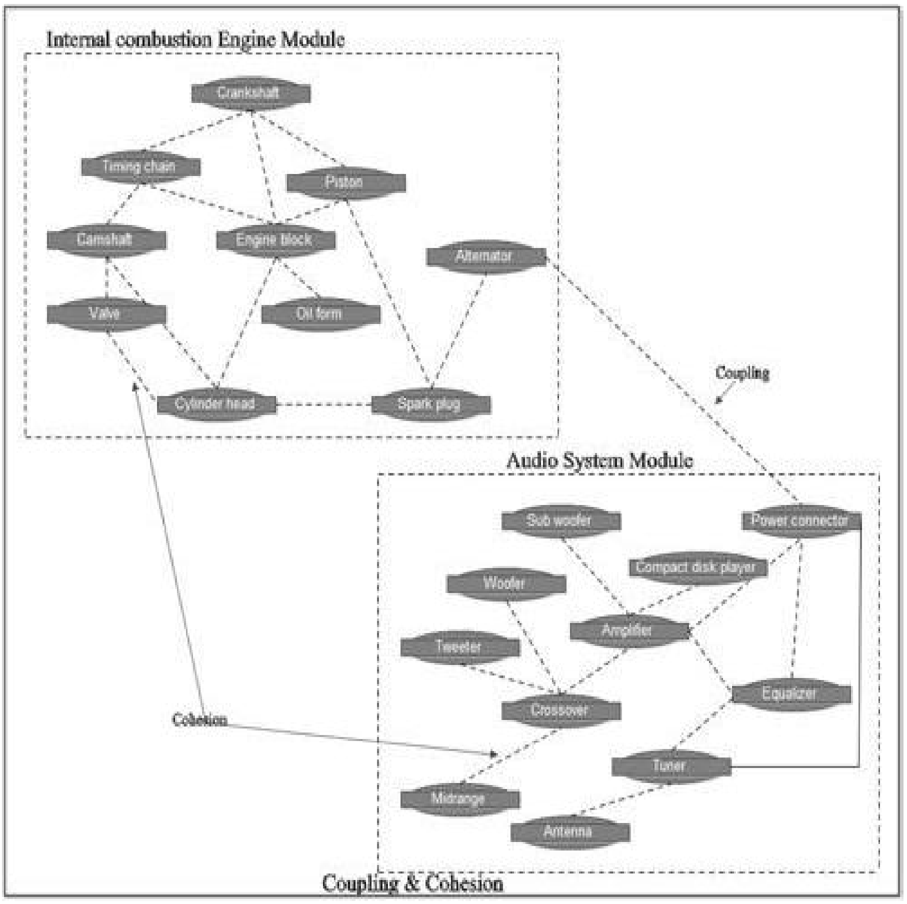
reusability of software design is the most valuable way of reducing the cost involved in software development.

2. **Ensure information hiding:** The benefits of modularity cannot be achieved merely by decomposing a program into several modules; rather each module should be designed and developed in such a way that the information hiding is ensured. It implies that the implementation details of one module should not be visible to other modules of the program. The concept of information hiding helps in reducing the cost of subsequent design changes.

Modularity has become an accepted approach in every engineering discipline. With the introduction of modular design, complexity of software design has considerably reduced; change in the program is facilitated that has encouraged parallel development of systems. To achieve effective modularity, design concepts like functional independence are considered to be very important.

## 8.4 Functional Independence

Functional independence is the refined form of the design concepts of modularity, abstraction, and information hiding. Functional independence is achieved by developing a module in such a way that it uniquely performs given sets of function without interacting with other parts of the system. The software that uses the property of functional independence is easier to develop because its functions can be categorized in a systematic manner. Moreover, independent modules require less maintenance and testing activity, as secondary effects caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is the key to a good software design and a good design results in high-quality software. There exist two qualitative criteria for measuring functional independence, namely, coupling and cohesion.



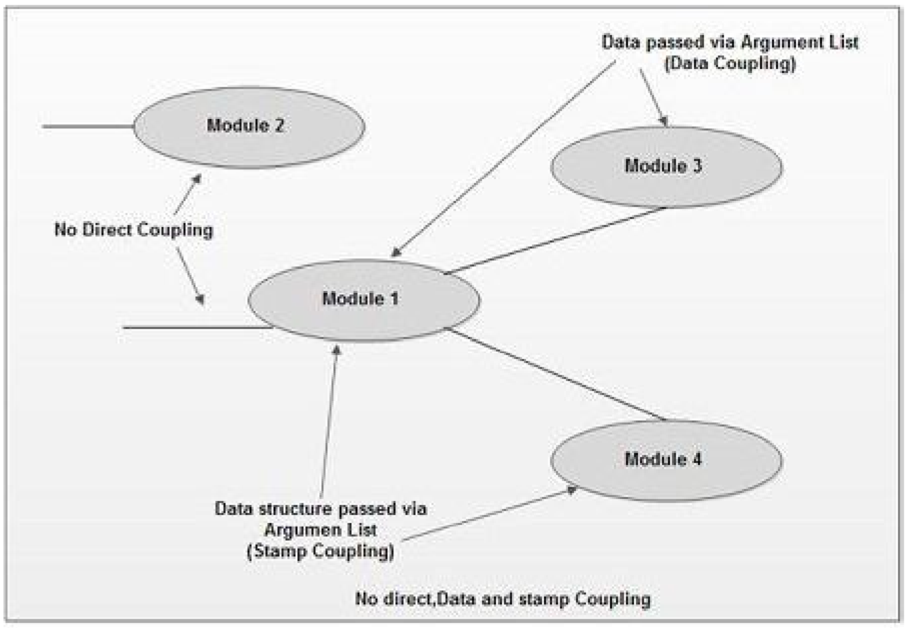
**COUPLING AND COHESION**

## 8.4.1 Coupling

Coupling measures the degree of interdependence among the modules. Several factors like interface complexity, type of data that pass across the interface, type of communication, number of interfaces per module, etc. influence the strength of coupling between two modules. For better interface and well-structured system, the modules should be loosely coupled in order to minimize the ‘ripple effect’ in which modifications in one module results in errors in other modules. Module coupling is categorized into the following types.

**1. No direct coupling**

Two modules are said to be ‘no direct coupled’ if they are independent of each other.



**REPRESENTATION OF NO DIRECT COUPLING**

2. **Data coupling**

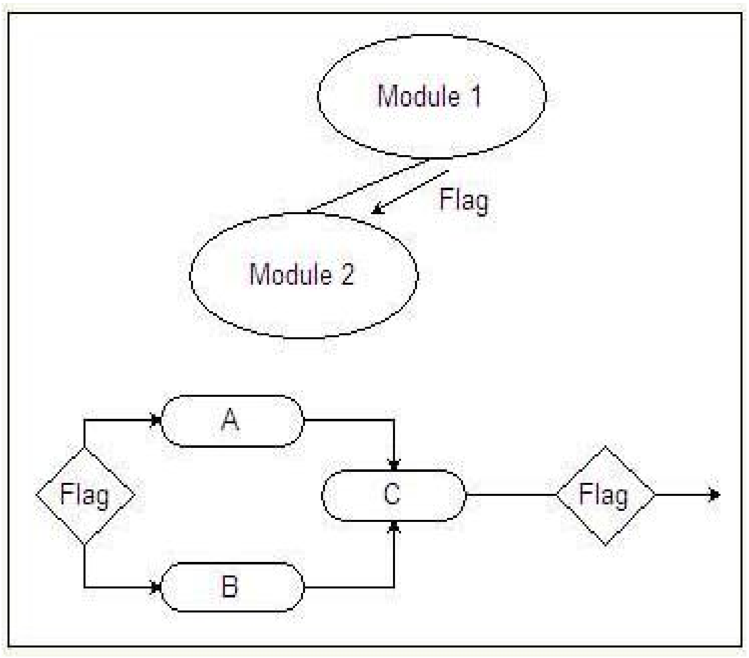
Two modules are said to be 'data coupled' if they use parameter list to pass data items for communication.

3. **Stamp coupling**

Two modules are said to be 'stamp coupled' if they communicate by passing a data structure that stores additional information than what is required to perform their functions.

4. **Control coupling**

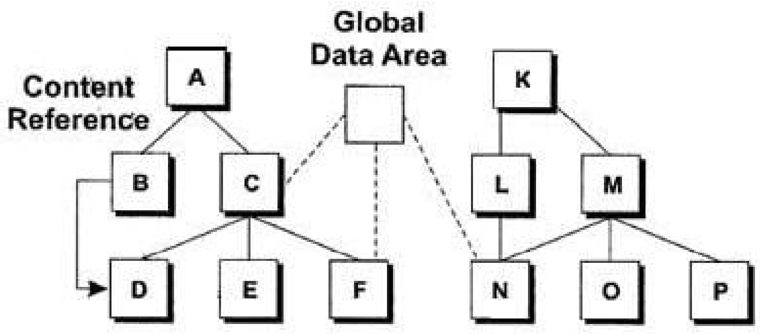
Two modules are said to be 'control coupled' if they communicate (pass a piece of information intended to control the internal logic) using at least one 'control flag'. The control flag is a variable whose value is used by the dependent modules to make decisions.



**FLOW DIAGRAM FOR CONTROL COUPLING**

**5. Content coupling**

Two modules are said to be ‘content coupled’ if one module modifies data of some other module or one module branches into the middle of another module.



Content and Common Coupling

**6. Common coupling**

Two modules are said to be ‘common coupled’ if *they* both reference a common data block.

## 8.4.2 Cohesion

Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules. The tighter the elements are bound to each other, the higher will be the cohesion of a module. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa.

Various types of cohesion are listed below.

1. **Functional cohesion**

In this, the elements within the modules are concerned with the execution of a single function.

2. **Sequential cohesion**

In this, the elements within the modules are involved in activities in such a way that the output from one activity becomes the input for the next activity.

3. **Communicational cohesion**

In this, the elements within the modules perform different functions, yet each function references the same input or output information.

4. **Procedural cohesion**

In this, the elements within the modules are involved in different and possibly unrelated activities.

5. **Temporal cohesion**

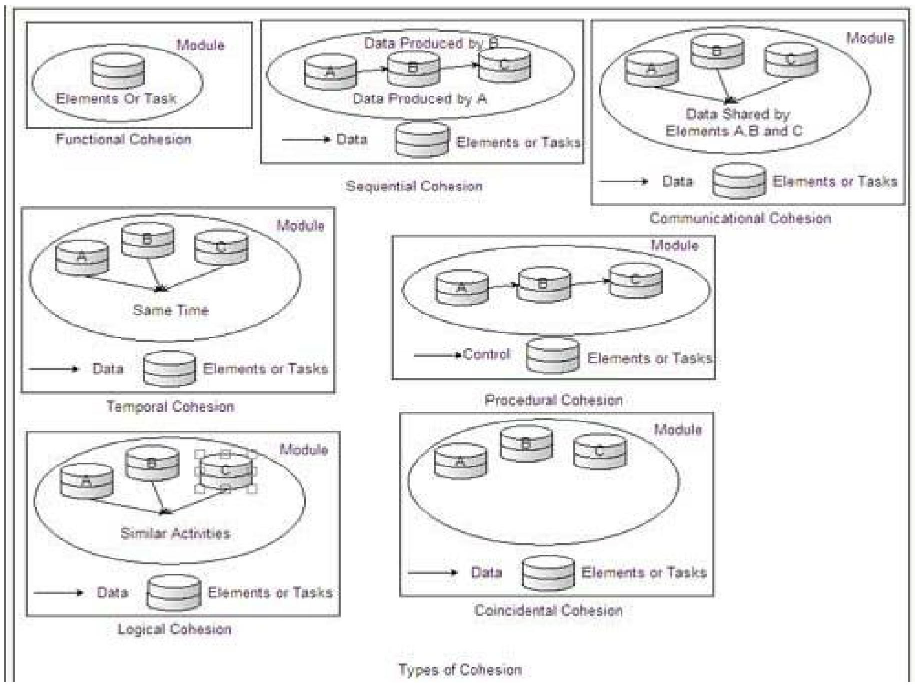
In this, the elements within the modules contain unrelated activities that can be carried out at the same time.

6. **Logical cohesion**

In this, the elements within the modules perform similar activities, which are executed from outside the module.

1. **Coincidental cohesion:** In this, the elements within the modules perform activities with no meaningful relationship to one another. Although these words were spoken many years ago, they remain true today. When the design model is translated into source code, we must follow a set of design principles that not only perform the translation but also do not "introduce bugs to start with."

It is possible to represent the component-level design using a programming language. In essence, the program is created using the design model as a guide. An alternative approach is to represent the procedural design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code. Regardless of the mechanism that is used to represent the component level design, the data structures, interfaces, and algorithms defined should conform to a variety of well-established procedural design guidelines that help us to avoid errors as the procedural design evolves. In this chapter, we examine these design guidelines.



**DIFFERENT TYPES OF COHESION**

## 8.5 Structured Programming

The foundations of component-level design were formed in the early 1960s and were solidified with the work of Edsgar Dijkstra and his colleagues ([BOH66], [DIJ65], [DIJ76]). In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized “maintenance of functional domain.” That is, each construct had a predictable logical structure, was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations. Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking.* To understand this process, consider the way in which you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line.

Understanding is enhanced when readily recognizable logical patterns are encountered. Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties..

## 8.6 Graphical Design Notation

“A picture is worth a thousand words,” but it’s rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

- A flowchart is quite simple pictorially. A box is used to indicate a processing step.
- A diamond represents a logical condition, and arrows show the flow of control.

- The *sequence* is represented as two processing boxes connected by an line (arrow) of control.
- *Condition,* also called *if -then-else,* is depicted as a decision diamond that if true, causes *then-part* processing to occur, and if false, invokes *else-part* processing.
- *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task repetitively as long as the condition holds true. A *repeat until* executes the loop task first, then tests a condition and repeats the task until the condition fails.
- The selection (or select-case) construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

The structured constructs may be nested within one another, repeat-until forms the then part of if-then-else (shown enclosed by the outer dashed boundary). Another if-then-else forms the else part of the larger condition. Finally, the condition itself becomes a second block in a sequence.

By nesting constructs in this manner, a complex logical schema may be developed. It should be noted that any one of the blocks could reference another module, thereby accomplishing procedural layering implied by program structure. In general, the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required.

More important, additional complication of all logical tests along the path of escape can cloud software control flow, increase the possibility of error, and have a negative impact on readability and maintainability. What can we do?

The designer is left with two options:

(1) The procedural representation is redesigned so that the “escape branch” is not required at a nested location in the flow of control or

(2) the structured constructs are violated in a controlled manner; that is, a constrained branch out of the nested flow is designed. Option 1 is obviously the ideal approach, but option 2 can be accommodated without violating of the spirit of structured programming.

Another graphical design tool, the *box diagram,* evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. Developed by Nassi and Shneiderman [NAS73] and extended by Chapin the diagrams (also called *Nassi-Shneiderman charts, N-S charts,* or *Chapin charts*) .It have the following characteristics:

(1) functional domain (that is, the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation,

(2) arbitrary transfer of control is impossible, (3) the scope of local and/or global data can be easily determined, (4) recursion is easy to represent.

The graphical representation of structured constructs using the box diagram. The fundamental element of the diagram is a box. To represent sequence, two boxes are connected bottom to top. To represent if-then-else, a condition box is followed by a then-part and else-part box. Repetition is depicted with a bounding pattern that encloses the process (do-while part or repeat-until part) to be repeated. Finally, selection is represented using the graphical form.

Like flowcharts, a box diagram is layered on multiple pages as processing elements of a module are refined. A “call” to a subordinate module can be represented within a box by specifying the module name enclosed by an oval.

## 8.7 Tabular Design Notation

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm. In a comprehensive treatment of this design tool, Ned Chapin states [HUR83]:

Some old software tools and techniques mesh well with new tools and techniques of software engineering. Decision tables are an excellent example. Decision tables preceded software engineering by nearly a decade, but fit so well with software engineering that they might have been designed for that purpose.

Decision table organization is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing *rule.*

The following steps are applied to develop a decision table:

**1.** List all actions that can be associated with a specific procedure (or module).

**2.** List all conditions (or decisions made) during execution of the procedure.

**3.** Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.

**4.** Define rules by indicating what action(s) occurs for a set of conditions.

To illustrate the use of a decision table, consider the following excerpt from a processing narrative for a public utility billing system:

If the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 KWH (kilowatt-hours). Otherwise, computer billing applies a Schedule A rate structure. However, if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 KWH, with additional consumption billed according to Schedule B.

Each of the five rules indicates one of five viable conditions (i.e., a T (true) in both fixed rate and variable rate account makes no sense in the context of this procedure; therefore, this condition is omitted). As a general rule, the decision table can be effectively used to supplement other procedural design notation.

## 8.8 Program Design Language

*Program design language* (PDL), also called *structured English* or *pseudocode,* is “a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)” [CAI75]. PDL is used as a generic reference for a design language. At first glance PDL looks like a modern programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled (at least not yet).

However, PDL tools currently exist to translate PDL into a programming language "skeleton" and/or a graphical representation (e.g., a flowchart) of design. These tools also produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.

A program design language may be a simple transposition of a language such as Ada or C. Alternatively, it may be a product purchased specifically for procedural design. Regardless of origin, a design language should have the following characteristics:

• A fixed syntax of keywords that provide for all structured constructs, data declaration, and modularity characteristics.

• A free syntax of natural language that describes processing features.

• Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures.

• Subprogram definition and calling techniques that support various modes of interface description.

A basic PDL syntax should include constructs for subprogram definition, interface description, data declaration, techniques for block structuring, condition constructs, repetition constructs, and I/O constructs. The format and semantics for some of these PDL constructs are presented in the section that follows.

It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, inter-process synchronization, and many other features. The application design for which PDL is to be used should dictate the final form for the design language.

## 8.9 Comparison of Design Notation

In the preceding section, we presented a number of different techniques for representing a procedural design. A comparison must be predicated on the premise that any notation for component-level design, if used correctly, can be an invaluable aid in the design process; conversely, even the best notation, if poorly applied, adds little to understanding. With this thought in mind, we examine criteria that may be applied to compare design notation.

Design notation should lead to a procedural representation that is easy to understand and review. In addition, the notation should enhance “code to” ability so that code does, in fact, become a natural by-product of design. Finally, the design representation must be easily maintainable so that design always represents the program correctly.

The following attributes of design notation have been established in the context of the general characteristics described previously:

**a. Modularity**

Design notation should support the development of modular software and provide a means for interface specification.

**b. Overall simplicity**

Design notation should be relatively simple to learn, relatively easy to use, and generally easy to read.

**c. Ease of editing**

The procedural design may require modification as the software process proceeds. The ease with which a design representation can be edited can help facilitate each software engineering task.

**d. Machine readability**

Notation that can be input directly into a computer-based development system offers significant benefits.

**e. Maintainability**

Software maintenance is the most costly phase of the software life cycle. Maintenance of the software configuration nearly always means maintenance of the procedural design representation.

**f. Structure enforcement**

The benefits of a design approach that uses structured programming concepts have already been discussed. Design notation that enforces the use of only the structured constructs promotes good design practice.

### g. Automatic processing

A procedural design contains information that can be processed to give the designer new or better insights into the correctness and quality of a design. Such insight can be enhanced with reports provided via software design tools.

### h. Data representation

The ability to represent local and global data is an essential element of component-level design. Ideally, design notation should represent such data directly.

### i. Logic verification

Automatic verification of design logic is a goal that is paramount during software testing. Notation that enhances the ability to verify logic greatly improves testing adequacy.

## Check your Progress

1. **Fill in the blanks** Component-level design, also called ____________.
2. ____________ implements processing steps that are essential in the specification of any algorithm
3. **Choose the best answer** : There exist two qualitative criteria for measuring functional independence, namely _______and ________.

   a. User interface and design b. coupling and correctiond cohesion and logic verification d. coupling and cohesion.
4. Program design language (PDL), also called structured English or ____________.

   a. pseudocode b. syntax code c. graphic notation d. design structure
5. **Say True or False** The ability to represent local and global data is not an essential element of component- level design.

## 8.10 User Interface

User interface is the front-end application view to which user interacts in order to use the software. The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interface screens

There are two types of User Interface:

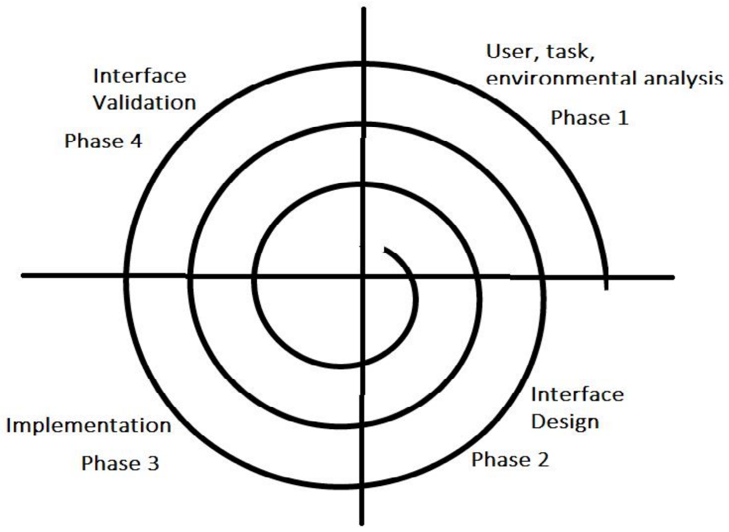
1. **Command Line Interface:** Command Line Interface provides a command prompt, where the user types the command and feeds to the system. The user needs to remember the syntax of the command and its use.
2. **Graphical User Interface:** Graphical User Interface provides the simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, user interprets the software.

## 8.10.1 User Interface Design Process

The analysis and design process of a user interface is iterative and can be represented by a spiral model. The analysis and design process of user interface consists of four framework activities.

**1. User, task, environmental analysis, and modeling:**

Initially, the focus is based on the profile of users who will interact with the system, i.e. understanding, skill and knowledge, type of user, etc., based on the user’s profile users are made into categories. From each category requirements are gathered. Based on the requirements developer understand how to develop the interface.



**Spiral Diagram**

Once all the requirements are gathered a detailed analysis is conducted. In the analysis part, the tasks that the user performs to establish the goals of the system are identified, described and elaborated. The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are:

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

## 2. Interface Design

The goal of this phase is to define the set of interface objects and actions i.e. Control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system. Specify the action sequence of tasks and subtasks, also called a user scenario. Indicate the state of the system when the user performs a particular task. Always follow the three golden rules stated by Theo Mandel. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. This phase serves as the foundation for the implementation phase.

**3. Interface construction and implementation**

The implementation activity begins with the creation of prototype (model) that enables usage scenarios to be evaluated. As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

**4. Interface Validation**

This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly and it should be able to handle a variety of tasks. It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

## 8.10.2 Golden Rules for Designing Interface

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface.

**a. Place the user in control**

- Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions: The user should be able to easily enter and exit the mode with little or no effort.
- Provide for flexible interaction: Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc., and hence all interaction mechanisms should be provided.
- Allow user interaction to be interruptible and undoable: When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.
- Streamline interaction as skill level advances and allow the interaction to be customized: Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.

- Hide technical internals from casual users: The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.

- Design for direct interaction with objects that appear on screen: The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

**b. Reduce the user's memory load**

Reduce demand on short-term memory: When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.

- Establish meaningful defaults: Always initial set of defaults should be provided to the average user, if a user needs to add some new features then he should be able to add the required features.

- Define shortcuts that are intuitive: Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.

- The visual layout of the interface should be based on a real-world metaphor: Anything you represent on a screen if it is a metaphor for real-world entity then users would easily understand.

- Disclose information in a progressive fashion: The interface should be organized hierarchically i.e. on the main screen the information about the task, an object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

**c. Make the interface consistent**

e. Allow the user to put the current task into a meaningful context: Many interfaces have dozens of screens. So it is important to provide indicators consistently so that the user know about the doing work. The user should also know from which page has navigated to the current page and from the current page where can navigate.

f. Maintain consistency across a family of applications: The development of some set of applications all should follow and implement the same design, rules so that consistency is maintained among applications.

g. If past interactive models have created user expectations do not make changes unless there is a compelling reason.

The overall process for designing a user interface begins with the creation of different models of system function (as perceived from the outside). The human- and computer-oriented tasks that are required to achieve system function are then delineated; design issues that apply to all interface designs are considered; tools are used to prototype and ultimately implement the design model; and the result is evaluated for quality.

## 8.11 Summary

The various component level design model have been read in this lesson. The concept about the different notation used during the design is also clearly described. Further explains in detail about the user interface design process.

## 8.12 Check Your Answers

1. procedural design
2. Sequence
3. Coupling and cohesion
4. pseudocode
5. False

## 8.13 Model Questions

1. What are the Steps involved for Component-Level Design ?
2. Write short notes on coupling and cohesion.
3. What are the steps applied to develop a decision table ?
4. Write briefly about Program Design Language.
5. List the golden rules for designing interface .

# LESSON-9

# INTERFACE DESIGN MODELS AND SOFTWARE CONFIGURATION MANAGEMENT (SCM)

## 9.1 Learning Objectives

Upon successful completion of this lesson, you will be able to:

- Design the user interface.
- Understand the different design models available in user interface.
- Learn the how to do interface process
- Evaluate the design process. - System Configuration Management (SCM)
- Understand about the version control and change management in software configuration management.

## Structure

## 9.2 Introduction

In this lesson, explains about the necessity of user interface during the software development. It provides guideline while designing the user interface. It discuss about the tools available for implementing the user interface. It further discuss about the software configuration management in detail.

## 9.3 Interface Design Models

Four different models come into play when a user interface is to be designed. The software engineer creates a design model, a human engineer (or the software engineer) establishes a user model, the end-user develops a mental image that is often called the user's model or the system perception, and the implementers of the system create a system image [RUB88]. Unfortunately, each of these models may differ significantly.

The role of interface designer is to reconcile these differences and derive a consistent representation of the interface. A design model of the entire system incorporates data, architectural, interface, and procedural representations of the software. The requirements specification may establish certain constraints that help to define the user of the system, but the interface design is often only incidental to the design model.

The user model establishes the profile of end-users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [SHN90]. In addition, users can be categorized as

• **Novices.** No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.

• **Knowledgeable, intermittent users.** Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

• **Knowledgeable, frequent users.** Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The system perception (user's model) is the image of the system that end-users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain.

A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

The system image combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describe system syntax and semantics.

When the system image and the system perception are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the system image must accurately reflect syntactic and semantic information about the interface.

## 9.4 The User Interface Design Process

The design process for user interfaces is iterative and can be represented using a spiral model. The user interface design process encompasses four distinct framework activities

**1.** User, task, and environment analysis and modeling

**2.** Interface design

**3.** Interface construction

**4.** Interface validation

The spiral implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the implementation activity involves prototyping—the only practical way to validate what has been designed.

The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, the software engineer attempts to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

• Where will the interface be located physically?

• Will the user be sitting, standing, or performing other tasks unrelated to the Interface?

• Does the interface hardware accommodate space, light, or noise constraints?

• Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.Validation focuses on:

(1) The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

(2) The degree to which the interface is easy to use and easy to learn;

(3) The users' acceptance of the interface as a useful tool in their work.

As we have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

## 9.5 Task Analysis And Modeling

Functional decomposition or stepwise refinement as a mechanism for refining the processing tasks that are required for software to accomplish some desired function.

Task analysis for interface design uses either an elaborative or object-oriented approach but applies this approach to human activities. Task analysis can be applied in two ways. As we have already noted, an interactive, computer-based system is often used to replace a manual or semi-manual activity.

To understand the tasks that must be performed to accomplish the goal of the activity, a human engineer must understand the tasks that humans currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, the human engineer can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

Regardless of the overall approach to task analysis, a human engineer must first define and classify tasks. We have already noted that one approach is stepwise elaboration. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. By observing an interior designer at work, the engineer notices that interior design comprises a number of major activities: furniture layout, fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping.

Each of these major tasks can be elaborated into subtasks. For example, furniture layout can be refined into the following tasks:

(1) Draw a floor plan based on room dimensions;

(2) Place windows and doors at appropriate locations;

(3) use furniture templates to draw scaled furniture outlines on floor plan;

(4) Move furniture outlines to get best placement;

(5) label all furniture outlines;

(6) draw dimensions to show location;

(7) Draw perspective view for customer.

A similar approach could be used for each of the other major tasks. Subtasks 1–7 can each be refined further. Subtasks 1–6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.

The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

An alternative approach to task analysis takes an object-oriented point of view. The human engineer observes the physical objects that are used by the interior designer and the actions that are applied to each object.

For example, the furniture template would be an object in this approach to task analysis. The interior designer would select the appropriate template, move it to a position on the floor plan, trace the furniture outline and so forth. The design model for the interface would not provide a literal implementation for each of these actions, but it would define user tasks that accomplish the end result (drawing furniture outlines on the floor plan).

Ideally, the individual had some training in human engineering and user interface design.

## 9.6 Interface Design Activities

Once task analysis has been completed, all tasks (or objects and actions) required by the end-user have been identified in detail and the interface design activity commences.

The first interface design steps [NOR86] can be accomplished using the following approach:

**1.** Establish the goals and intentions for each task.

**2.** Map each goal and intention to a sequence of specific actions.

**3.** Specify the action sequence of tasks and subtasks, also called a user scenario, as it will be executed at the interface level.

**4.** Indicate the state of the system; that is, what does the interface look like at the time that a user scenario is performed?

**5.** Define control mechanisms; that is, the objects and actions available to the user to alter the system state.

**6.** Show how control mechanisms affect the state of the system.

**7.** Indicate how the user interprets the state of the system from information provided through the interface.

### 9.6.1 Defining Interface Objects and Actions

An important step in interface design is the definition of interface objects and the actions that are applied to them. That is, a description of a user scenario is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon). The implication of this action is to create a hard-copy report. An application object represents application-specific data that is not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing.

The list itself might be sorted, merged, or purged (menu-based actions) but it is not dragged and dropped via user interaction. Goals include a consideration of the usefulness of the task, its effectiveness in accomplishing the overriding business objective, the degree to which the task can be learned quickly, and the degree to which users will be satisfied with the ultimate implementation of the task.

When the designer is satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items is conducted. If a real world metaphor is appropriate for the application, it is specified at this time and the layout is organized in a manner that complements the metaphor.

## 9.6.2 Design Issues

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command Labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and customer frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristics: length and variability. If the length of system response is too long, user frustration and stress is the inevitable result. However, a very brief response time can also be detrimental if the user is being paced by the interface. A rapid response may force the user to rush and therefore, make mistakes.

Variability refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command is preferable to a response that varies from 0.1 to 2.5 seconds. The user is always off balance, always wondering whether something "different" has occurred behind the scenes.

Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of “user manuals” may be the only option. In many cases, however, modern software provides on-line help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

Two different types of help facilities are encountered: integrated and add-on. An integrated help facility is designed into the software from the beginning. It is often context sensitive, enabling the user to select from those topics that are relevant to the actions currently being performed. Obviously, this reduces the time required for the user to obtain help and increases the “friendliness” of the interface.

An add-on help facility is added to the software after the system has been built. In many ways, it is really an on-line user’s manual with limited query capability. The user may have to search through a list of hundreds of topics to find appropriate guidance, often making many false starts and receiving much irrelevant information. There is little doubt that the integrated help facility is preferable to the add-on approach.

A number of design issues must be addressed when a help facility is considered:

• Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.

• How will the user request help? Options include a help menu, a special function key, or a HELP command.

• How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.

• How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.

• How will help information be structured? Options include a “flat” structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

Error messages and warnings are “bad news” delivered to users of interactive systems when something has gone away. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.

There are few computer users who have not encountered an error of the form:

**SEVERE SYSTEM FAILURE — 14A**

Somewhere, an explanation for error 14A must exist; otherwise, why would the designers have added the identification? Yet, the error message provides no real indication of what is wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.

In general, every error message or warning produced by an interactive system Should have the following characteristics:

• The message should describe the problem in jargon that the user can understand.

• The message should provide constructive advice for recovering from the error.

• The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

• The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the “error color.”

• The message should be “nonjudgmental.” That is, the wording should never place blame on the user.

Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point and pick interfaces has reduced reliance on typed commands, but many power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands are provided as a mode of interaction:

• Will every menu option have a corresponding command?

• What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.

• How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?

• Can commands be customized or abbreviated by the user?

## 9.7 Implementation Tools

Once a design model is created, it is implemented as a prototype, examined by users (who fit the user model described earlier) and modified based on their comments. To accommodate this iterative design approach, a broad class of interface design and prototyping tools has evolved. Called user-interface toolkits or user-interface development systems (UIDS), these tools provide components or objects that facilitate creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment.

Using prepackaged software components to create a user interface, a UIDS provides Built- in mechanisms for:

- managing input devices (such as a mouse or keyboard)
- validating user input
- handling errors and displaying error messages
- providing feedback (e.g., automatic input echo)
- providing help and prompts

It should be noted that in some cases (e.g., aircraft cockpit displays) the first step might be to simulate the interface on a display device rather than prototyping it.

• handling windows and fields, scrolling within windows

• establishing connections between application software and the interface

• insulating the application from interface management functions

• allowing the user to customize the interface

These functions can be implemented using either a language-based or graphical approach.

## 9.8 Design Evaluation

Once an operational user interface prototype has been created, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal “test drive,” in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.

The user interface evaluation cycle takes the form shown in Figure 15.3. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files).

Design modifications are made based on user input and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built? If potential problems can be uncovered and corrected early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

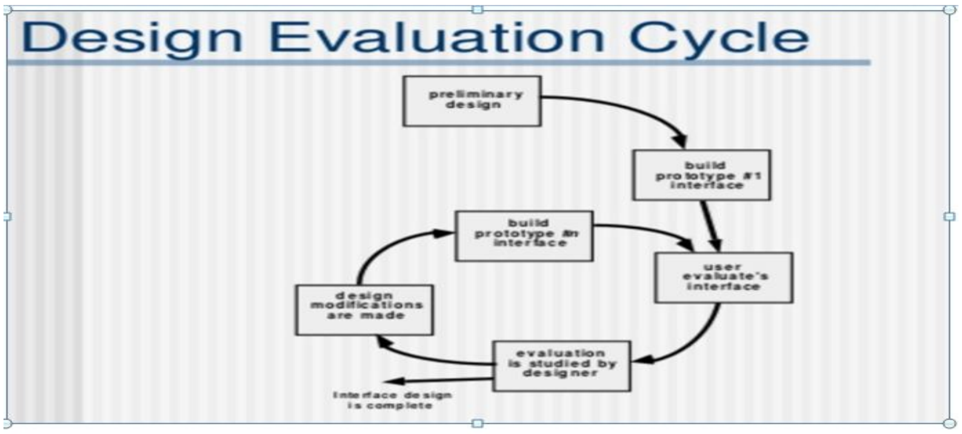

Design Evaluation Cycle
preliminary design
build prototype #1 interface
build prototype #n interface
user evaluate's interface
design modifications are made
evaluation is studied by designer
interface design is complete

Figure 9.1 Design Evaluation Cycle

**1.** The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.

**2.** The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.

**3.** The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.

**4.** Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype.

Questions can be all :

(1) simple yes/no response

(2) numeric response

(3) scaled (subjective) response, or

(4) percentage (subjective) response.

Examples are:

**1.** Were the icons self-explanatory? If not, which icons were unclear?

**2.** Were the actions easy to remember and to invoke?

**3.** How many different actions did you use?

**4.** How easy was it to learn basic system operations (scale 1 to 5)?

**5.** Compared to other interfaces you've used, how would this rate—top 1%, top 10%, top 25%, top 50%, bottom 50%?

If quantitative data are desired, a form of time study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user's ability to tap the computational power of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces:

(1) Place the user in control,

(2) Reduce the user's memory load, and

(3) Make the interface consistent.

To achieve an interface that abides by these principles, an organized design process must be conducted. User interface design begins with the identification of user, task, and environmental requirements. Task analysis is a design activity that defines user tasks and actions using either an elaborative or object-oriented approach.

Once tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. This provides a basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user.

The user interface is the window into the software. In many cases, the interface molds a user's perception of the quality of the system. If the "window" is smudged, wavy, or broken, the user may reject an otherwise powerful computer-based system.

### Check your Progress

1. **Fill in the blanks** The design process for user interfaces is iterative and can be represented using a ________ model.
2. An alternative approach to task analysis takes an __________point of view.
3. **Choose the correct answers:**System response time is the primary complaint for many _____ applications.

   a. Design   b. interactive   c. pilot   d. web
4. The message should provide _________advice for recovering from the error.

   a. Destructive   b. friendly  c. user friendly   d. constructive
5. **Say True or False** User-Interface development systems (UIDS), does not provide Built- in mechanisms for providing help and prompts.

## 9.9 System Configuration Management (SCM)

It is an arrangement of exercises which controls change by recognizing the items for change, setting up connections between those things, making/characterizing instruments for overseeing diverse variants, controlling the changes being executed in the current framework, inspecting and revealing/reporting on the changes made. It is essential to control the changes in light of the fact that if the changes are not checked legitimately then they may wind up undermining a well-run programming. In this way, SCM is a fundamental piece of all project management activities.

### 9.9.1 Tasks in SCM process

**1. Configuration Identification**

Configuration identification is a method of determining the scope of the software system. With the help of this step, you can manage or control something even if you don’t know what it is. It is a description that contains the CSCI type (Computer Software Configuration Item), a project identifier and version information.

**Activities during this process:**

- Identification of configuration Items like source code modules, test case, and requirements specification.
- Identification of each CSCI in the SCM repository, by using an object-oriented approach
- The process starts with basic objects which are grouped into aggregate objects. Details of what, why, when and by whom changes in the test are made
- Every object has its own features that identify its name that is explicit to all other objects
- List of resources required such as the document, the file, tools, etc.

**Example:** Instead of naming a File login.php it should be named login_v1.2.php where v1.2 stands for the version number of the file Instead of naming folder “Code” it should be named “Code_D” where D represents code should be backed up daily.

**2. Baseline**

A baseline is a formally accepted version of a software configuration item. It is designated and fixed at a specific time while conducting the SCM process. It can only be changed through formal change control procedures.

**Activities during this process:**

- Facilitate construction of various versions of an application
- Defining and determining mechanisms for managing various versions of these work products
- The functional baseline corresponds to the reviewed system requirements
- Widely used baselines include functional, developmental, and product baselines In simple words, baseline means ready for release.

**3. Change Control**

Change control is a procedural method which ensures quality and consistency when changes are made in the configuration object. In this step, the change request is submitted to software configuration manager.

**Activities during this process:**

- Control ad-hoc change to build stable software development environment. Changes are committed to the repository
- The request will be checked based on the technical merit, possible side effects and overall impact on other configuration objects.
- It manages changes and making configuration items available during the software lifecycle

**4. Configuration Status Accounting**

Configuration status accounting tracks each release during the SCM process. This stage involves tracking what each version has and the changes that lead to this version.

**Activities during this process:**

- Keeps a record of all the changes made to the previous baseline to reach a new baseline
- Identify all items to define the software configuration
- Monitor status of change requests
- Complete listing of all changes since the last baseline
- Allows tracking of progress to next baseline
- Allows to check previous releases/versions to be extracted for testing

**5. Configuration Audits and Reviews**

Software Configuration audits verify that all the software product satisfies the baseline needs. It ensures that what is built is what is delivered.

**Activities during this process:**

- Configuration auditing is conducted by auditors by checking that defined processes are being followed and ensuring that the SCM goals are satisfied.
- To verify compliance with configuration control standards. auditing and reporting the changes made

- SCM audits also ensure that traceability is maintained during the process.
- Ensures that changes made to a baseline comply with the configuration status reports
- Validation of completeness and consistency

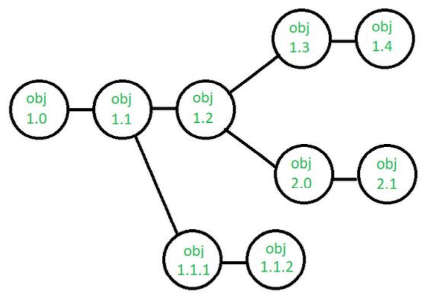
## 9.9.2 Processes involved in SCM

Configuration management provides a disciplined environment for smooth control of work products. It involves the following activities:

1. **Identification and Establishment:** Identifying the configuration items from products that compose baselines at given points in time (a baseline is a set of mutually consistent Configuration Items, which has been formally reviewed and agreed upon, and serves as the basis of further development). Establishing relationship among items, creating a mechanism to manage multiple level of control and procedure for change management system.

2. **Version control**

Creating versions/specifications of the existing product to build new products from the help of SCM system. A description of version is given below:



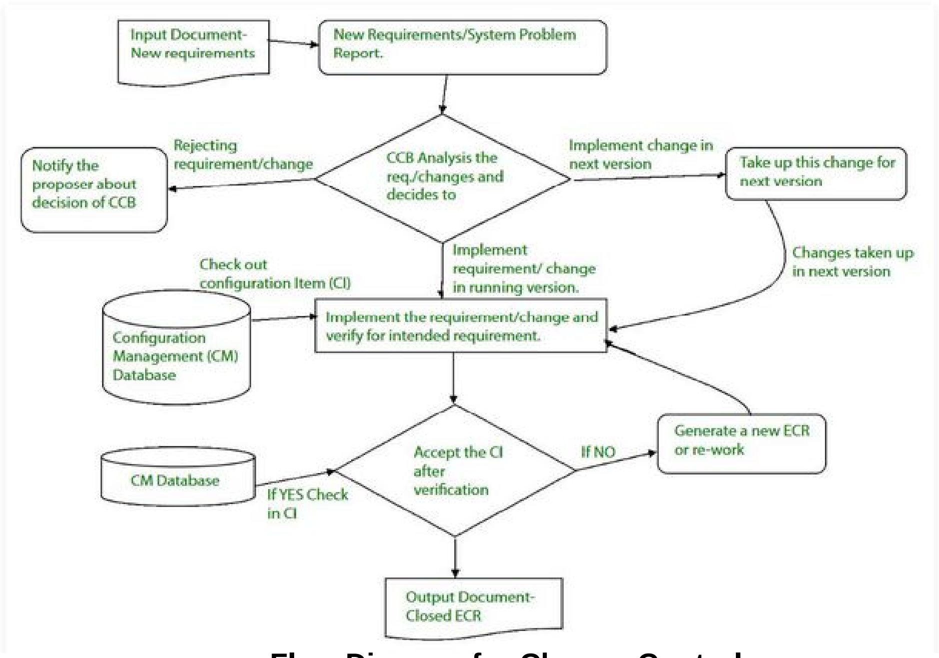
**Diagrammatic Description of Version Control**

Suppose after some changes, the version of configuration object changes from 1.0 to 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The development of object 1.0 continues through 1.3 and 1.4, but finally, a noteworthy change to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.

- Combines procedures and tools to manage the different versions of configuration objects created during the software process

- Version control systems require the following capabilities
  - o Project repository – stores all relevant configuration objects
  - o Version management capability – stores all versions of a configuration object (enables any version to be built from past versions)
  - o Make facility – enables collection of all relevant configuration objects and construct a specific software version
  - o Issues (bug) tracking capability – enables team to record and track status of outstanding issues for each configuration object
  - Uses a system modeling approach (template – includes component hierarchy and component build order, construction rules, verification rules)

3. **Change control:** Controlling changes to Configuration items (CI). The change control process is explained in Figure below. A change request (CR) is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a change report, which is used by a change control board (CCB) —a person or group who makes a final decision on the status and priority of the change. An engineering change Request (ECR) is generated for each approved change.


Input Document-
New requirements
New Requirements/System Problem Report.
Rejecting requirement/change
CCB Analysis the req./changes and decides to
Implement change in next version
Notify the proposer about decision of CCB
Take up this change for next version
Check out configuration Item (CI)
Implement requirement/ change in running version.
Changes taken up in next version
Configuration Management (CM) Database
Implement the requirement/change and verify for intended requirement.
Generate a new ECR or re-work
Accept the CI after verification
If NO
CM Database
If YES Check in CI
Output Document-
Closed ECR

**Flow Diagram for Change Control**

Also CCB notifies the developer in case the change is rejected with proper reason. The ECR describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is “checked out” of the project database, the change is made, and then the object is tested again. The object is then “checked in” to the database and appropriate version control mechanisms are used to create the next version of the software.

- Change request is submitted and evaluated to assess technical merit and impact on the other configuration objects and budget
- Change report contains the results of the evaluation
- Change control authority (CCA) makes the final decision on the status and priority of the change based on the change report
- Engineering change order (ECO) is generated for each change approved (describes change, lists the constraints, and criteria for review and audit)
- Object to be changed is checked-out of the project database subject to access control parameters for the object
- Modified object is subjected to appropriate SQA and testing procedures
- Modified object is checked-in to the project database and version control mechanisms are used to create the next version of the software
- Synchronization control is used to ensure that parallel changes made by different people don’t overwrite one another

## 4. Configuration Auditing

A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified. The audit confirms the completeness, correctness and consistency of items in the SCM system and track action items from the audit to closure.

A software audit is an independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria. Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities. Consequently, each audit must be carefully planned. An audit can require a number of individuals to perform a variety of tasks over a fairly short period of time. Tools to support the planning and conduct of an audit can greatly facilitate the process. Software configuration auditing determines the extent to which an item satisfies the required functional and physical characteristics. Informal audits of this type can be conducted at key points in the life cycle.

Two types of formal audits might be required by the governing contract (for example, in contracts covering critical software):

- Functional Configuration Audit (FCA) and
- Physical Configuration Audit (PCA).

Successful completion of these audits can be a prerequisite for the establishment of the product baseline.

**Software Functional Configuration Audit:** The purpose of the software FCA is to ensure that the audited software item is consistent with its governing specifications. The output of the software verification and validation activities (see Verification and Validation in the Software Quality KA) is a key input to this audit.

**Software Physical Configuration Audit:** The purpose of the software physical configuration audit (PCA) is to ensure that the design and reference documentation is consistent with the as-built software product.

**In-Process Audits of a Software Baseline:** As mentioned above, audits can be carried out during the development process to investigate the current status of specific elements of the configuration. In this case, an audit could be applied to sampled baseline items to ensure that performance is consistent with specifications or to ensure that evolving documentation continues to be consistent with the developing baseline item.

## 5. Reporting

Providing accurate status and current configuration data to developers, tester, end users, customers and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guide etc. .

## 9.10 Summary

The various design model have been read in this lesson. The concept about the data design is also clearly described. The various types of architecture styles are learnt with the steps involved.  Further you will know how to refine the architectural design. It discuss about the software configuration management and its version control and change management in detail.

## 9.11 Check Your Answers

1. Spiral
2. Object-oriented
3. Interactive
4. Constructive
5. False

## 9.12 Model Questions

1. List the **golden rules for designing interface .**
2. write short notes on task analysis and modeling in the interface design.
3. How to define Interface Objects and Actions in the user interface design?
4. What are the  important principles to guide the design of effective user interfaces ?
5. Write short notes on Software configuration management .

# LESSON-10

# SOFTWARE METRICS

## 10.1 Learning Objective

Upon successful completion of this lesson, you will be able to:

- define what are the 4 P's in Software Project Management
- describe the responsibilities of Software Project Manager
- Understand the W5HH Principle.

### Structure

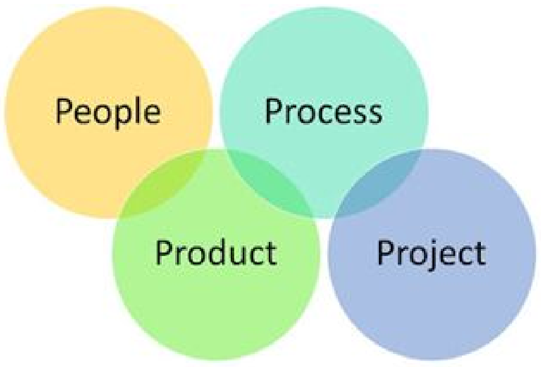
## 10.2 Introduction

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management.

## 10.3 The Management Spectrum

In software engineering, the management spectrum describes the management of a software project. The management of a software project starts from requirement analysis and finishes based on the nature of the product, it may or may not end because almost all software products faces changes and requires support. It is about turning the project from plan to reality.

A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the product.



**The Four P's of Management Spectrum**

### 10.3.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a people management capability maturity model (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability".

People of a project includes from manager to developer, from customer to end user. But mainly people of a project highlight the developers. It is so important to have highly skilled and motivated developers that the Software Engineering Institute has developed a People Management Capability Maturity Model (PM-CMM). Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The people management maturity model deûnes the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

## Roles and Responsibilities of People

The responsibilities of people deal with the cultivation of motivated, highly skilled people and the different roles consist of the stakeholders, the team leaders, and the software team.

### The Stakeholders

Five categories of stakeholders

- **Senior managers**– define business issues that often have significant influence on the project
- **Project (technical) managers** – plan, motivate, organize, and control the practitioners who do the work
- **Practitioners** – deliver the technical skills that are necessary to engineer a product or application
- **Customers** – specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome
- **End users** – interact with the software once it is released for production use

### Team Leaders

- Competent practitioners often fail to make good team leaders; they just don’t have the right people skills
- Qualities to look for in a team leader
  - Motivation – the ability to encourage technical people to produce to their best ability
  - Organization – the ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product

  - Ideas or innovation – the ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application

- Team leaders should use a problem-solving management style
  - Concentrate on understanding the problem to be solved
  - Manage the flow of ideas
  - Let everyone on the team know, by words and actions, that quality counts and that it will not be compromised

**The Software Team**

Seven project factors to consider when structuring a software development team

- The difficulty of the problem to be solved
- The size of the resultant program(s) in source lines of code
- The time that the team will stay together
- The degree to which the problem can be modularized
- The required quality and reliability of the system to be built
- The rigidity of the delivery date
- The degree of sociability (communication) required for the project

## 10.3.2 The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered and technical and management constraints should be identiûed. Without this information, it is impossible to deûne reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

The product is the ultimate goal of the project. This is any types of software product that has to be developed. To develop a software product successfully, all the product objectives and scopes should be established, alternative solutions should be considered and technical and management constraints should be identified beforehand. Lack of these information, it is impossible to define reasonable and accurate estimation of the cost, an effective assessment of risks, a realistic breakdown of project tasks or a manageable project schedule that provides a meaningful indication of progress.

The software developer and customer must meet to deûne product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements analysis .Objectives identify the overall goals for the product(from the customer’s point of view) without considering how these goals will be achieved. Scope identifies the primary data, functions and behaviors that characterize the product and more importantly attempt to bind these characteristics in a quantitative manner.

### 10.3.3 The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets- tasks, milestones, work product and quality assurance points-enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software conûguration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

A software process provides the framework from which a comprehensive plan for software development can be established. A number of different tasks sets— tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities overlay the software process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

### 10.3.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, we still struggle. In 1998, industry data indicated that 26 percent of software projects failed outright and 46 percent experienced cost and schedule overruns. Although the success rate for software projects has improved somewhat, our project failure rate remains higher than it should be. In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring and controlling the project.

## 10.4 Responsibility of Software Project Manager

Software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised. Another view of the characteristics that define an effective project manager emphasizes four key traits:

**Problem solving**

An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain ûexible enough to change direction if initial attempts at problem solution are fruitless.

**Managerial Identity**

A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

**Achievement**

To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.

**Influence and Team Building**

An effective project manager must be able to “read” people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

## 10.5 W5HH Principle

Barry Boehm states: “you need an organizing principle that scales down to provide simple plans for simple projects.” Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the WWWWWHH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan.

Boehm’s W5HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide an excellent planning outline for the project manager and the software team.

Created by software engineer Barry Boehm, the purpose behind the **W5HH principles** is to work through the objectives of a software project, the project timeline, team member responsibilities, management styles and necessary resources. In an article he wrote on the topic, Boehm stated that an organizing principle is needed that works for any size project. So, he developed W5HH as a guiding principle for software projects.

The W5HH principle may have a funny-sounding name, but it too is designed for practicality. Each letter in W5HH stands for one in a series of questions to help a project manager lead. (Notice there is five ‘’W” questions and two ‘’H” questions).

### 10.5.1 Example - Why the W5HH?

Bonnie and her colleague, Clyde, are software project managers for the fictitious company, Software-R-Us. Bonnie works in the business side of software applications while Clyde’s work is focused on entertainment-based products. They’ve each been tasked with bringing a new software product to market in the third quarter of the year and, while both have smart teams, good ideas and big plans, one project manager is more successful in developing an end product.

### 10.5.2 Definition of the W5HH Model

**Why is the system being developed?** The answer to this question enables all parties to assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

**What will be done, by when?** The answers to these questions help the team to establish a project schedule by identifying key project tasks and the milestones that are required by the customer.

**Who is responsible for a function?** The answer to this question helps accomplish this.

**Where they are organizationally located?** Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

**How will the job be done technically and managerially?** Once product scope is established, a management and technical strategy for the project must be defined.

**How much of each resource is needed?** The answer to this question is derivedby developing estimates based on answers to earlier questions.

| **W5HH** | **The Question** | **What It Means** |
|---|---|---|
| Why? | Why is the system being developed? | This focuses a team on the business reasons for developing the software. |
| What? | What will be done? | This is the guiding principle in determining the tasks that need to be completed. |
| When? | When will it be completed? | This includes important milestones and the timeline for the project. |
| Who? | Who is responsible for each function? | This is where you determine which team member takes on which responsibilities. You may also identify external stakeholders with a claim in the project. |
| Where? | Where are they organizationally located? | This step gives you time to determine what other stakeholders have a role in the project and where they are found. |
| How? | How will the job be done technically and managerially? | In this step, a strategy for developing the software and managing the project is concluded upon. |
| How Much? | How much of each resource is needed? | The goal of this step is to figure out the amount of resources necessary to complete the project. |

**The W5HH Principle (or $W^5$HHWH)**

- Why is this system being developed?  defines business motive
- What will be done?  determines s/w function

- When will it be done?  establishes schedule
- Who is responsible for what?  identifies roles
- Where they are organizationally located?  clarifies relationships
- How will the job be accomplished?  addresses technical and managerial details
- How much of each resource is needed?  initiates estimates
- What can go wrong? considers risks
- How do I know things are going right? tracks and controls

## 10.6 Process and Project Metrics

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. The four reasons for measuring software are processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

### 10.6.1 Process Metrics

- These are the metrics pertaining to the Process Quality. They measure efficiency and effectiveness of various processes.

1. **Cost of Quality:** It is a measure in terms of money for the quality performance within an organization Cost of quality = (review + testing + verification review + verification testing + QA + configuration management + measurement + training + rework review + rework testing)/ total effort x 100
2. **Cost of poor Quality:** It is the cost of implementing imperfect processes and products. Cost of poor quality = rework effort/ total effort x 100
3. **Defect Density:** It is the number of defects detected in the software during the development divided by the size of the software (typically in KLOC or FP) Defect density for a project = Total number of defects/ project size in KLOC or FP

4. **Review Efficiency:** defined as the efficiency in harnessing/ detecting review defects in the verification stage. Review efficiency = (number of defects caught in review)/ total number of defects caught) x 100

5. **Testing Efficiency:** Testing efficiency = 1 – ((defects found in acceptance)/ total no of testing defects) x 100

6. **Defect Removal Efficiency:** Gives the efficiency with which defects were detected and minimum defects were filtered down to the customer. Defect removal efficiency = (1 – (total defects caught by customer/ total no of defects)) x 100

### 10.6.2 Project Metrics

- These are the metrics pertaining to the Project Quality. They measure defects, cost, schedule, productivity and estimation of various project resources and deliverables.

1. **Schedule Variance:** Any difference between the scheduled completion of an activity and the actual completion is known as Schedule Variance. Schedule variance = ((Actual calendar days – Planned calendar days) + Start variance)/ Planned calendar days x 100.

2. **Effort Variance:** Difference between the planned outlined effort and the effort required to actually undertake the task is called Effort variance. Effort variance = (Actual Effort – Planned Effort)/ Planned Effort x 100.

3. **Size Variance:** Difference between the estimated size of the project and the actual size of the project (normally in KLOC or FP) Size variance = (Actual size – Estimated size)/ Estimated size x 100.

4. **Requirement Stability Index:** Provides visibility and understanding into the magnitude and impact of requirements changes. RSI = 1- ((No of changed + No of deleted + No of added) / Total no of Initial requirements) x100.

5. **Productivity (Project):** It is a measure of output from a related process for a unit of input. Project Productivity = Actual Project Size / Actual Effort spent for the project.

6. **Productivity (for test case preparation):** Productivity in test case preparation = Actual no of test cases/ actual effort spent on test case preparation

7. **Productivity (for test case execution):** Productivity in test case execution = actual number of test cases / actual effort spend on testing.

8. **Productivity (defect detection):** Productivity in defect detection = Actual number of defects (review + testing) / actual effort spent on (review + testing)

9. **Productivity (defect fixation):** Productivity in defect fixation = actual no of defects fixed/ actual effort spent on defect fixation

10. **Schedule variance for a phase:** The deviation between planned and actual schedule for the phases within a project. Schedule variance for a phase = (Actual Calendar days for a phase – Planned calendar days for a phase + Start variance for a phase)/ (Planned calendar days for a phase) x 100

11. **Effort variance for a phase:** The deviation between planned and actual effort for various phases within the project. Effort variance for a phase = (Actual effort for a phase – planned effort for a phase)/ (planned effort for a phase) x 100

## 10.6.3 Measures, Metrics, and Indicators

Measure - provides a quantitative indication of the size of some product or process attribute. Measurement - is the act of obtaining a measure. Metric - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

### Process and Project Indicators

Metrics should be collected so that process and product indicators can be ascertained Process indicators enable software project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality.

### Process Metrics

Private process metrics (e.g. defect rates by individual or module) are known only to the individual or team concerned. Public process metrics enable organizations to make strategic changes to improve the software process. Metrics should not be used to evaluate the performance of individuals. Statistical software process improvement helps and organization to discover where they are strong and where they are week.

## Project Metrics

Software project metrics are used by the software team to adapt project workflow and technical activities. Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis. Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

## Software Measurement

Direct measures of software engineering process include cost and effort.

Direct measures of the product include lines of code (LOC), execution speed, memory size, defects per reporting time period. Indirect measures examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

## Size-Oriented Metrics

Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC. Size oriented metrics are widely used but their validity and applicability is widely debated.

## Function-Oriented Metrics

Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity. Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes. Feature points and 3D function points provide a means of extending the function point concept to allow its use with real-time and other engineering applications.

The relationship of LOC and function points depends on the language used to implement the software.

## Software Quality Metrics

Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification). Software quality factors requiring measures include correctness (defects per KLOC), maintainability (mean time to change), integrity (threat and security), and usability (easy to learn, easy to use, productivity increase, user attitude).

Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied throughout the process framework.

**Integrating Metrics with Software Process**

Many software developers do not collect measures. Without measurement it is impossible to determine whether a process is improving or not. Baseline metrics data should be collected from a large, representative sampling of past software projects. Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

**Statistical Process Control**

It is important to determine whether the metrics collected are statistically valid and not the result of noise in the data.Control charts provide a means for determining whether changes in the metrics data are meaningful or not.Zone rules identify conditions that indicate out of control processes (expressed as distance from mean in standard deviation units).

**Metrics for Small Organizations**

Most software organizations have fewer than 20 software engineers.

Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect. Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

## Check your Progress

1. I am a measure of the filtering ability of the quality assurance and control activities. Who am I?

2. Difference between the estimated size of the project and the actual size of the project is __________.

3. List out the qualities of a Team Leader.

4. Differentiate Private and Public Process Metrics.

5. Expand LOC.

## 10.7 Software Measurement

Software measurement is a quantified attribute (see also: measurement) of a characteristic of a software product or the software process. It is a discipline within software engineering. The process of software measurement is defined and governed by ISO Standard ISO 15939 (software measurement process).

### 10.7.1 Software metrics

**Software size, functional measurement:** The primary measure of software is size, specifically functional size. The generic principles of functional size are described in the ISO/IEC 14143. Software size is principally measured in function points. It can also be measured in lines of code, or specifically, source lines of code (SLOC) which is functional code excluding comments. Whilst measuring SLOC is interesting, it is more an indication of effort than functionality. Two developers could approach a functional challenge using different techniques, and one might need only write a few lines of code, and the other might need to write many times more lines to achieve the same functionality. The most reliable method for measuring software size is code agnostic, from the user's point of view - in function points. The only ISO standards for measuring software size are COSMIC Functional Sizing and Function point.

**Measuring code:** One method of software measurement is metrics that are analyzed against the code itself. These are called software metrics and including simple metrics, such as counting the number of lines in a single file, the number of files in an application, the number of functions in a file, etc. Such measurements have become a common software development practice.

**Measuring software complexity, cohesion and coupling:** There are also more detailed metrics that help measure things like software complexity, Halstead, cohesion, and coupling.

### 10.7.2 Software Complexity

**Programming complexity** (or **software complexity**) is a term that encompasses numerous properties of a piece of software, all of which affect internal interactions. According to several commentators, there is a distinction between the terms complex and complicated. Complicated implies being difficult to understand but with time and effort, ultimately knowable. Complex, on the other hand, describes the interactions between a numbers of entities. As the number of entities increases, the number of interactions between them would increase exponentially, and it would get to a point where it would be impossible to know and understand all of them. Similarly, higher levels of complexity in software increases the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes. In more extreme cases, it can make modifying the software virtually impossible. The idea of linking software complexity to the maintainability of the software has been explored extensively by Professor Manny Lehman, who developed his Laws of Software Evolution from his research. He and his co-Author Les Belady explored numerous possible Software Metrics in their oft cited book that could be used to measure the state of the software, eventually reaching the conclusion that the only practical solution would be to use one that uses deterministic complexity models.

**Halstead**

**Halstead complexity measures** are software metrics introduced by Maurice Howard Halstead in 1977 as part of his treatise on establishing an empirical science of software development. Halstead made the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code.

Halstead's goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (analogous to the gas equation). Thus his metrics are actually not just complexity metrics.

## 10.7.3 Cohesion

In computer programming, **cohesion** refers to the degree to which the elements inside a module belong together. In one sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class. In another sense, it is a measure of the strength of relationship between the class's methods and data themselves.

Cohesion is an ordinal type of measurement and is usually described as "high cohesion" or "low cohesion". Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

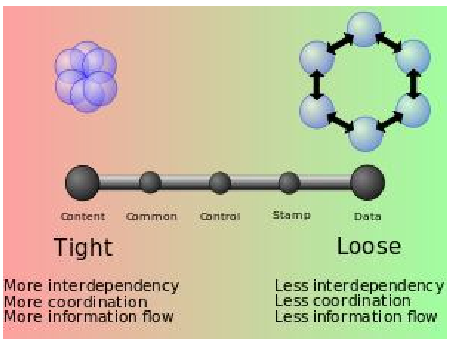
Cohesion is often contrasted with coupling, a different concept. High cohesion often correlates with loose coupling, and vice versa. The software metrics of coupling and cohesion were invented by Larry Constantine in the late 1960s as part of Structured Design, based on characteristics of “good” programming practices that reduced maintenance and modification costs. Structured Design, cohesion and coupling were published in the article Stevens, Myers & Constantine (1974) and the book Yourdon & Constantine (1979); the latter two subsequently became standard terms in software engineering.

## 10.7.4 Coupling

In software engineering, coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are the strength of the relationships between modules. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

The software quality metrics of coupling and cohesion were invented by Larry Constantine in the late 1960s as part of a structured design, based on characteristics of “good” programming practices that reduced maintenance and modification costs. Structured design, including cohesion and coupling, were published in the article Stevens, Myers & Constantine (1974) and the book Yourdon & Constantine (1979), and the latter subsequently became standard terms.

**Types of coupling**



**Tight and Loosely Coupled**

Coupling can be “low” (also “loose” and “weak”) or “high” (also “tight” and “strong”). Some types of coupling, in order of highest to lowest coupling, are as follows:

**Procedural programming**

A module here refers to a subroutine of any kind, i.e. a set of one or more statements having a name and preferably its own set of variable names.

**Content coupling (high)**

Content coupling is said to occur when one module uses the code of other module, for instance a branch. This violates information hiding - a basic design concept.

**Common coupling**

Common coupling is said to occur when several modules have access to the same global data. But it can lead to uncontrolled error propagation and unforeseen side-effects when changes are made.

**External coupling**

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

**Control coupling**

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

**Stamp coupling (data-structured coupling)**

Stamp coupling occurs when modules share a composite data structure and use only parts of it, possibly different parts (e.g., passing a whole record to a function that needs only one field of it). In this situation, a modification in a field that a module does not need may lead to changing the way the module reads the record.

**Data coupling**

Data coupling occurs when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

**Subclass coupling**

It describes the relationship between a child and its parent. The child is connected to its parent, but the parent is not connected to the child.

**Temporal coupling**

It occurs when two actions are bundled together into one module just because they happen to occur at the same time.

**Dynamic coupling**

The goal of this type of coupling is to provide a run-time evaluation of a software system. It has been argued that static coupling metrics lose precision when dealing with an intensive use of dynamic binding or inheritance .

**Semantic coupling**

This kind of coupling considers the conceptual similarities between software entities using, for example, comments and identifiers and relying on techniques such as Latent Semantic Indexing (LSI).

**Logical coupling**

Logical coupling (or evolutionary coupling or change coupling) exploits the release history of a software system to find change patterns among modules or classes: e.g., entities that are likely to be changed together or sequences of changes (a change in a class A is always followed by a change in a class B).

## 10.8 Summary

Software measurement is a diverse collection of these activities that range from models predicting software project costs at a specific stage to measures of program structure. Measure can be defined as quantitative indication of amount, dimension, capacity, or size of product and process attributes. Measurement can be defined as the process of determining the measure. Metrics can be defined as quantitative measures that allow software engineers to identify the efficiency and improve the quality of software process, project, and product.

## 10.9 Check Your Answers

1. Defect removal efficiency (DRE).
2. Size Variance.
3. Motivation, Organization and Ideas or innovation.
4. Private process metrics are known only to the individual or team concerned. Public process metrics are known to all people in the organization.
5. Line of Code.

## 10.10 Model Questions

1. What is Schedule Variance?
2. Define Coupling.
3. Define Software Measurement.
4. Explain the four P's of Management Spectrum.
5. What is the W5HH principle?. Explain in detail.
6. Describe in detail about the coupling and cohesion Software Metric.
7. Explain about the Process and Product Metrics and Indicators.

# LESSON -11

# SOFTWARE PROJECT ESTIMATION

## 11.1 Learning Objective

Upon successful completion of this lesson, you will be able to:

- Define how to estimate the project (techniques),
- Estimate how much time it will require to complete the project (Schedule),
- Evaluate who will be doing the project (resources),
- Define what is the budget required to deliver the project (cost), and
- Identify any intermediary dependencies that may delay or impact the project (Risks).

### Structure

## 11.2 Introduction

Estimation techniques are of utmost importance in software development life cycle, where the time required to complete a particular task is estimated before a project begins. Estimation is the process of finding an estimate, or approximation, which is a value that can be used for some purpose even if input data may be incomplete, uncertain, or unstable.

## 11.3 Software Project Estimation

There are many different types of estimation techniques used in project management with various streams as Engineering, IT, Construction, Agriculture, Accounting etc. A Project manager is often challenged to align mainly six project constraints - Scope, Time, Cost, Quality, Resources and Risk in order to accurately estimate the project. The software project estimation is inherently a predictive activity; **estimation will always be fraught with inaccuracies.** Another problem with inaccurate estimates is that the critical path identified might not actually be the critical path, which in turn, can't be consoled by impeccable risk management.

There are 3 major parts to project estimation mainly:-

- Effort estimation,
- Cost estimation, and
- Resource estimate

While accurate estimates are the basis of sound project planning, there are many techniques used as project management best practices in estimation as - Analogous estimation, Parametric estimation, Delphi method, 3 Point Estimate, Expert Judgment, Published Data Estimates, Vendor Bid Analysis, Reserve Analysis, Bottom Up Analysis, and Simulation. Usually, during the early stages of a project life cycle, the project requirements are feebly known and less information is available to estimate the project. The initial estimate is drawn merely by assumptions knowing the scope at a high level, this is known as 'Ball-park estimates', a term very often used by project managers. Estimation is the process of finding an estimate, or approximation, which is a value that can be used for some purpose even if input data may be incomplete, uncertain, or unstable. Estimation determines how much money, effort, resources, and time it will take to build a specific system or product. Estimation is based on "

- Past Data/Past Experience
- Available Documents/Knowledge
- Assumptions
- Identified Risks

The four basic steps in Software Project Estimation are “

- Estimate the size of the development product.
- Estimate the effort in person-months or person-hours.
- Estimate the schedule in calendar months.
- Estimate the project cost in agreed currency.

**Observations on Estimation**

- Estimation need not be a one-time task in a project. It can take place during “
    - Acquiring a Project.
    - Planning the Project.
    - Execution of the Project as the need arises.
- Project scope must be understood before the estimation process begins. It will be helpful to have historical Project Data.
- Project metrics can provide a historical perspective and valuable input for generation of quantitative estimates.
- Planning requires technical managers and the software team to make an initial commitment as it leads to responsibility and accountability.
- Past experience can aid greatly.
- Use at least two estimation techniques to arrive at the estimates and reconcile the resulting values. Refer Decomposition Techniques in the next section to learn about reconciling estimates.

- Plans should be iterative and allow adjustments as time passes and more details are known.

**Estimation Accuracy**

Accuracy is an indication of how close something is to reality. Whenever you generate an estimate, everyone wants to know how close the numbers are to reality. You will want every estimate to be as accurate as possible, given the data you have at the time you generate it. And of course you don't want to present an estimate in a way that inspires a false sense of confidence in the numbers.

Important factors that affect the accuracy of estimates are “

- The accuracy of the entire estimate's input data.
- The accuracy of any estimate calculation.
- How closely the historical data or industry data used to calibrate the model matches the project you are estimating.
- The predictability of your organization's software development process.
- The stability of both the product requirements and the environment that supports the software engineering effort.
- Whether or not the actual project was carefully planned, monitored and controlled, and no major surprises occurred that caused unexpected delays.

Following are some guidelines for achieving reliable estimates “

- Base estimates on similar projects that have already been completed.
- Use relatively simple decomposition techniques to generate project cost and effort estimates.
- Use one or more empirical estimation models for software cost and effort estimation.

**Estimation Issues**

Often, project managers resort to estimating schedules skipping to estimate size. This may be because of the timelines set by the top management or the marketing team. However, whatever the reason, if this is done, and then at a later stage it would be difficult to estimate the schedules to accommodate the scope changes. While estimating, certain assumptions may be made. It is important to note all these assumptions in the estimation sheet, as some still do not document assumptions in estimation sheets.

Even good estimates have inherent assumptions, risks, and uncertainty, and yet they are often treated as though they are accurate.The best way of expressing estimates is as a range of possible outcomes by saying, for example, that the project will take 5 to 7 months instead of stating it will be complete on a particular date or it will be complete in a fixed no. of months. Beware of committing to a range that is too narrow as that is equivalent to committing to a definite date.

- You could also include uncertainty as an accompanying probability value. For example, there is a 90% probability that the project will complete on or before a definite date.
- Organizations do not collect accurate project data. Since the accuracy of the estimates depends on the historical data, it would be an issue.
- For any project, there is a shortest possible schedule that will allow you to include the required functionality and produce quality output. If there is a schedule constraint by management and/or client, you could negotiate on the scope and functionality to be delivered.
- Agree with the client on handling scope creeps to avoid schedule overruns.
- Failure in accommodating contingency in the final estimate causes issues. For e.g., meetings, organizational events.
- Resource utilization should be considered as less than 80%. This is because the resources would be productive only for 80% of their time. If you assign resources at more than 80% utilization, there is bound to be slippages.

**Estimation Guidelines**

One should keep the following guidelines in mind while estimating a project “

- During estimation, ask other people’s experiences. Also, put your own experiences at task.

- Assume resources will be productive for only 80 percent of their time. Hence, during estimation take the resource utilization as less than 80%.
- Resources working on multiple projects take longer to complete tasks because of the time lost switching between them.
- Include management time in any estimate.
- Always build in contingency for problem solving, meetings and other unexpected events.
- Allow enough time to do a proper project estimate. Rushed estimates are inaccurate, high-risk estimates. For large development projects, the estimation step should really be regarded as a mini project.
- Where possible, use documented data from your organization's similar past projects. It will result in the most accurate estimate. If your organization has not kept historical data, now is a good time to start collecting it.
- Use developer-based estimates, as the estimates prepared by people other than those who will do the work will be less accurate.
- Use several different people to estimate and use several different estimation techniques.
- Reconcile the estimates. Observe the convergence or spread among the estimates. Convergence means that you have got a good estimate. Wideband-Delphi technique can be used to gather and discuss estimates using a group of people, the intention being to produce an accurate, unbiased estimate.
- Re-estimate the project several times throughout its life cycle.

## 11.4 Project Estimation Techniques

**Top-down estimate:** Once more detail is learned on the scope of project, this technique is usually followed where high level chunks at the feature or design level is estimated and is decomposed progressively into smaller chunks or work-packets as information is detailed.

**Bottom-up estimate:** This technique is used when the requirements are known at a discrete level where the smaller work pieces are then aggregated to estimate the entire project. This is usually used when the information is only known in smaller pieces.

**Analogous estimating:** This technique is used when there is a reference of similar project executed and it is easy to correlate with other projects. Expert judgement and historical information of similar activities in referenced project is gathered to arrive at an estimate of the project.

**Parametric estimate:** This technique uses independent measurable variables from the project work. For example, the cost for construction of a building is calculated based on the smallest variable as the cost to build a square feet area, effort required to build a work packet is calculated from the variable as lines of codes in a software development project. This technique gives more accuracy in project estimation.

**Three point estimating:** This technique uses a mathematical approach as the weighted average of optimistic, most likely and pessimistic estimate of the work package. This is often known as the PERT (Program Evaluation and Review Technique).

**What if analysis:** This technique uses assumptions based on varying factors as scope, time, cost, resources etc to evaluate the possible outcomes of the project by doing an impact analysis. In a usual scenario, the project estimate is done by conducting estimation workshops with the stakeholders of the project, senior team members who could give valuable inputs to the estimation exercise. The high level scope is broken down into smaller work packages, components and activities, each work package is estimated by effort and resource needed to complete the work package. The project may be detailed into the smallest chunk that can be measured.

## 11.4.1 Types of Project Estimation Techniques

Project manager can estimate the listed factors using two broadly recognized techniques –

**Decomposition Technique**

This technique assumes the software as a product of various compositions. There are two main models -

- **Line of Code** Estimation is done on behalf of number of line of codes in the software product.
- **Function Points** Estimation is done on behalf of number of function points in the software product.

**Empirical Estimation Technique**

This technique uses empirically derived formulae to make estimation. These formulae are based on LOC or FPs.

- **Putnam Model**

  This model is made by Lawrence H. Putnam, which is based on Norden's frequency distribution (Rayleigh curve). Putnam model maps time and efforts required with software size.

- **COCOMO**

  COCOMO stands for Constructive Cost Model, developed by Barry W. Boehm. It divides the software product into three categories of software: organic, semi-detached and embedded.

## 11.5 Challenges in Project Estimation

In traditional software project estimation, the Scrum Master, Project Manager, or the Architect are responsible for producing estimates. Project activities are then allocated and project team members start working on the tasks or stories. As the developers start the work, they realize the initial software project estimate was wrong. If the push-back culture doesn't exist in the organization, developers will deliver something, regardless of the quality, usually by burning the midnight oil. Everybody is happy on meeting the deadline…until the client sends a stinker! The team members that worked on the task will be taken to the cleaners, and so will the Project Manager.

The problem with this scenario is that **no one asked the people who are in-charge of doing the task for their estimates**. If they did, then maybe the extended hours would have been avoided. Maybe, the client would have got her money's worth.

Another reason for project software estimation goes wrong is that resources are fixed, but a deadline is set. **The Triangle of Constraints states that in a project there are three constraints, Scope, Time, and Cost; if one of the constraints changes, then so will the others.** In this case, if the deadline is set, time is fixed. Therefore, either Scope has to be reduced or the cost needs to go up or maybe both scope and cost will change.

### 11.5.1 Best Practices

Mentioned below are techniques that'll help your estimates stick:

- **People who do the work should participate in the project estimation**: This is especially true when you estimate at the task level. In Agile projects, everyone must participate. This ensures greater ownership. It is required for the Agile team's DNA.
- **Bring in the experts**: Expert judgment is critical to project success. Therefore, if required, hire consultants. However, take care not to follow their estimates blindly. Leverage the team.
- **Estimate large pieces as a range**: Your project estimate for cooking rice will be more accurate than the project estimate for cooking a meal for 10 people. This is because the latter is more complex and comprised of several tasks. Estimates get more accurate when the task is small. For activities that have several tasks, you should use a range. For example, cooking a meal for 10 will take between 8 to 12 hours. This best practice is especially useful early on in the project when you need to create a high-level plan.
- **Use the Delphi Technique**: This is a method that helps converge opinions of the team should there be a conflict in software project estimation.

## 11.6 Decomposition Techniques

The Project Estimation Approach that is widely used is **Decomposition Technique**. Decomposition techniques take a divide and conquer approach. Size, Effort and Cost estimation are performed in a stepwise manner by breaking down a Project into major Functions or related Software Engineering Activities.

Software project estimation is a form of problem solving and in most cases, the problem to be solved (i.e developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose the problem, re-characterizing it as a set a smaller (and hopefully, more manageable) problems. Before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its size.

**Step 1** ” Understand the scope of the software to be built.

**Step 2** d Generate an estimate of the software size.

- Start with the statement of scope.
- Decompose the software into functions that can each be estimated individually.
- Calculate the size of each function.
- Derive effort and cost estimates by applying the size values to your baseline productivity metrics.
- Combine function estimates to produce an overall estimate for the entire project.

**Step 3** d Generate an estimate of the effort and cost. You can arrive at the effort and cost estimates by breaking down a project into related software engineering activities.

- Identify the sequence of activities that need to be performed for the project to be completed.
- Divide activities into tasks that can be measured.
- Estimate the effort (in person hours/days) required to complete each task.
- Combine effort estimates of tasks of activity to produce an estimate for the activity.
- Obtain cost units (i.e., cost/unit effort) for each activity from the database.
- Compute the total effort and cost for each activity.
- Combine effort and cost estimates for each activity to produce an overall effort and cost estimate for the entire project.

**Step 4** ” Reconcile estimates: Compare the resulting values from Step 3 to those obtained from Step 2. If both sets of estimates agree, then your numbers are highly reliable. Otherwise, if widely divergent estimates occur conduct further investigation concerning whether “

- The scope of the project is not adequately understood or has been misinterpreted.
- The function and/or activity breakdown is not accurate.
- Historical data used for the estimation techniques is inappropriate for the application, or obsolete, or has been misapplied.

**Step 5** ” Determine the cause of divergence and then reconcile the estimates.

The decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its “size.”

## 11.6.1 Software Sizing

The accuracy of a software project estimate is predicated on a number of things: **(1)**the degree to which the planner has properly estimated the size of the product to be built; **(2)** the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); **(3)** the degree to which the project plan reflects the abilities of the software team; and **(4)** the stability of product requirements and the environment that supports the software engineering effort.

Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing represents the project planner’s first major challenge. In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in LOC. If an indirect approach is chosen, size is represented as FP.

**Putnam and Myers suggest four different approaches to the sizing problem:**

- **“Fuzzy logic” sizing:** This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range. Although personal experience can be used, the planner should also have access to a historical database of projects8 so that estimates can be compared to actual experience.

- **Function point sizing:** The planner develops estimates of the information domain characteristics.

- **Standard component sizing:** Software is composed of a number of different “standard components” that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component. To illustrate, consider an information systems application. The planner estimates that 18 reports will be generated. Historical data indicates that 967 lines of COBOL are required per report. This enables the planner to estimate that 17,000 LOC will be required for the reports component. Similar estimates and computation are made for other standard components, and a combined size value (adjusted statistically) results.

- **Change sizing:** This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, and deleting code) of modifications that must be accomplished. Using an “effort ratio” for each type of change, the size of the change may be estimated.

### Check your Progress

1. Expand CoCoMo
2. ____________ uses a mathematical approach as the weighted average of optimistic.
3. Which of the following approach uses the approximate reasoning techniques?

- Change sizing (ii) Function point sizing (iii) Standard Component Sizing (iv) Fuzzy Logic sizing

4. ________is an indication of how close something is to reality.

5. List out the project constraints.

## 11.7 Project Scheduling

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and then arrange them keeping various factors in mind. They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated.

**Basic Principles of Project Scheduling:** Software project scheduling is an activity that distributes estimated effort across the planed project duration by allocating the effort to specific software engineering tasks. First, a macroscopic schedule is developed. a detailed schedule is redefined for each entry in the macroscopic schedule. A schedule evolves over time.

Basic principles guide software project scheduling:

- Compartmentalization
- Interdependency
- Time allocation
- Effort allocation
- Effort validation
- Defined responsibilities
- Defined outcomes
- Defined milestones

For scheduling a project, it is necessary to -

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

A project is made up of many tasks, and each task is given a start and end (or due date), so it can be completed on time. Likewise, people have different schedules, and their availability and vacation or leave dates need to be documented in order to successfully plan those tasks. Whereas people in the past might have printed calendars on a shared wall in the water-cooler room, or shared spreadsheets via email, today most teams use online project scheduling tools. Typically, project scheduling is just one feature within a larger project management software solution, and there are many different places in the software where scheduling takes place.

For example, most tools have task lists, which enable the manager to schedule multiple tasks, their due dates, sometimes the planned effort against that task, and then assign that task to a person. The software might also have resource scheduling, basically the ability to schedule the team's availability, but also the availability of non-human resources like machines or buildings or meeting rooms.

Because projects have so many moving parts, and are frequently changing, project scheduling software automatically updates tasks that are dependent on one another, when one scheduled task is not completed on time. It also generates automated email alerts, so team members know when their scheduled tasks are due or overdue, and to let the manager know when someone's availability has changed.

Project scheduling is simple when managed online, thankfully, especially since the software does all the hard part for you!

**How to Schedule a Project**

Before going deeper into project scheduling, let's review the fundamentals to project scheduling. Project scheduling occurs during the planning phase of the project. You have to ask yourself three questions to start:

1. What needs to be done?
2. When will it be done?
3. Who will do it?

Once you've got answers to these questions, then you can begin to plan dates, link activities, set the duration, milestones and resources. The following are the steps needed to schedule a project:

**Define Activities:** What are the activities that you have to do in the project? By using a Work Breakdown Structure (WBS) and a deliverables diagram, you can begin to take these activities and organize them by mapping out the tasks necessary to complete them in an order than makes sense.

**Do Estimates:** Now that you have the activities defined and broken down into tasks, you next have to determine the time and effort it will take to complete them. This is an essential piece of the equation in order to calculate the correct schedule.

**Determine Dependencies:** Tasks are not an island, and often one cannot be started until the other is completed. That's called a task dependency, and your schedule is going to have to reflect these linked tasks. One way to do this is by putting a bit of slack in your schedule to accommodate these related tasks.

**Assign Resources:** The last step to finalizing your planned schedule is to decide on what resources you are going to need to get those tasks done on time. You're going to have to assemble a team, and their time will need to be scheduled just like the tasks.

**How to Maintain Your Schedule Once the Project Is Initiated:** Once you've got all the pieces of your schedule together, the last thing you want to do is manually punch it into a static document like an Excel spreadsheet. Project management software can automate much of the process for you. But not all project management software is the same. There are programs on the market that are great for simple scheduling duties, but when you're leading a project, big or small, you need a tool that can adapt to the variety of scheduling issues you're going to need to track. Like noted above, there are three tiers of scheduling: tasks, people and projects.

**Scheduling Tasks:** What you want when scheduling tasks is not a glorified to-do list, but smart software that gives you the flexibility to handle the variety of responsibilities attached to each tasks in your project.

An interactive Gantt chart is crucial. You can add tasks and dates into your Gantt chart to have a visual representation of each task's duration. Better still, as dates change—as they inevitably do—you can simply drag and drop those changes and the whole Gantt chart is updated instantly. There's also automating processes to help with efficiencies. Email notifications are a great way to know immediately when a team member has completed a task. When they update, you know because your software is online and responding in real-time.

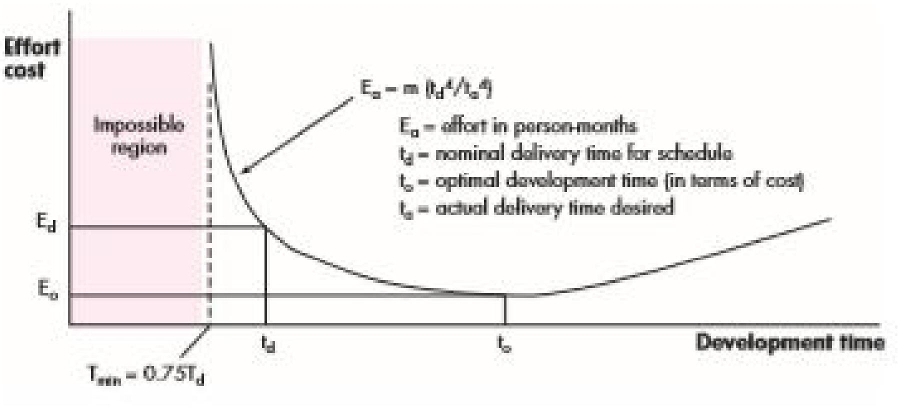
Continuing with automation, it's one way to scheduling tasks more efficiently. If there are recurring tasks on a project, they can be scheduled in your PM tool so that once set you don't have to worry about scheduling the same task over and over again.

**Scheduling People:** The tasks aren't going to complete themselves. That's why a team has been assembled, but if that team isn't scheduled the way you have carefully scheduled your task list, then you're not managing your project.Over the course of a project's lifecycle team members are going to take off for holidays, personal days or vacation. If you're not prepared for these times, and have scheduled other team members to pick up the slack in their absence, your schedule will suffer.

Integrating your calendar into project management software is a simple way to stay on top of your resources. There’s no reason to use a standalone calendar that sends you to another application every time you need to check on a team member’s availability.

Another way to stay on top of your scheduling is by integrating your task scheduling view on the Gantt chart with resource and workload scheduling features. You can schedule your team’s workload through color-coding, so you know at-a-glance who is behind, ahead or on schedule with their tasks.

In small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.



**Grant Chart**

## Scheduling Projects

We’re gone from the task level to the resources level of scheduling, but there’s no law that says you’re not working on a portfolio of projects. How can you keep on scheduling when juggling so many balls in the air at once?

The project dashboard is your best friend, whether you’re working on one or many projects. The dashboard is collecting all the real-time data collected by you and your teams, and then it’s organizing it according to any number of metrics to show you a picture of where you stand in real-time on the project or many projects. With a project dashboard you can note where tasks are being blocked and immediately adjust your schedule to resolve delays before they become a problem. You can also use the graphs and charts the dashboard automatically generates to drill down deeper and filter or customize the results to get the information you need, when you need it.

And that's just a fraction of what we could say about project scheduling. Our ongoing series explains and explores new and relevant terms in project management, focusing on a specific definition and summarizing what it means for anyone leading a project. But to really get to know scheduling, it's best to dive in with a project tool, your tasks and your team and create a new project schedule today.

## 11.8 Summary

Project scheduling is a mechanism to communicate what tasks need to get done and which organizational resources will be allocated to complete those tasks in what timeframe. A project schedule is a document collecting all the work needed to deliver the project on time. But when it comes to creating a project schedule, well, that's something few have deep experience with.

## 11.9 Check Your Answers

1. Constructive Cost Model

2. Three point estimating

3. "Fuzzy logic" sizing

4. Accuracy

5. Project Constraints- Scope, Time, Cost, Quality, Resources and Risk

## 11.10 Model Questions

1. Define Decomposition Technique.

2. Differentiate between top down and bottom up project estimation techniques.

3. Explain the project estimation techniques.

4. Describe the steps in decomposition technique.

5. List out the basic principles of project scheduling.

6. Explain about the Putnam and Myers approaches to the sizing problem.

# LESSON-12

# RISK MANAGEMENT

## 12.1 Learning Objective

Upon successful completion of this lesson, you will be able to:

- Describe what is risk management
- Describe about the types of risk in software project management
- Understand the methods of risk assessment

### Structure

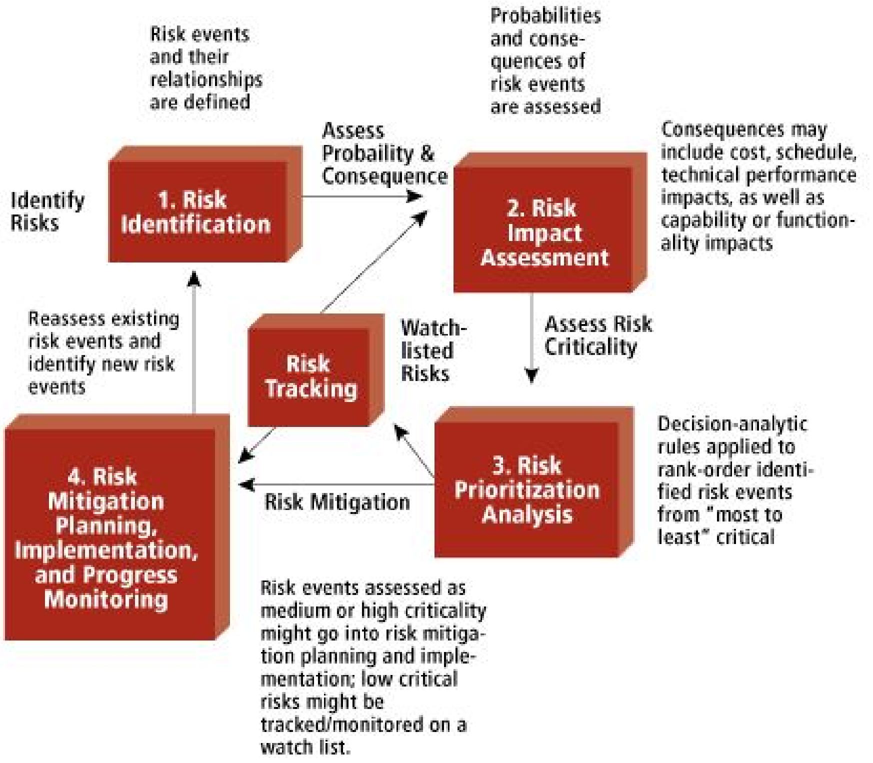
## 12.2 Introduction

The goal of most software development and software engineering projects is to be distinctive—often through new features, more efficiency, or exploiting advancements in software engineering. Any software project executive will agree that the pursuit of such opportunities cannot move forward without risk.

## 12.3 Risk Management

Software development is activity that uses a variety of technological advancements and requires high levels of knowledge. Because of these and other factors, every software development project contains elements of uncertainty. This is known as project risk. The success of a software development project depends quite heavily on the amount of risk that corresponds to each project activity. As a project manager, it's not enough to merely be aware of the risks. To achieve a successful outcome, project leadership must identify, assess, prioritize, and manage all of the major risks.

Because risks are painfully real and quite prevalent on all software projects, it's critically necessary that stakeholders work hard to identify, understand, and mitigate any risks that might threaten the success of a project. For projects that have time and cost constraints, our experience shows most clearly that successful software development efforts are those in which risk mitigation is a central management activity.



Risk events and their relationships are defined
Probabilities and consequences of risk events are assessed
Identify Risks
1. Risk Identification
Assess Probaility & Consequence
2. Risk Impact Assessment
Consequences may include cost, schedule, technical performance impacts, as well as capability or functionality impacts
Reassess existing risk events and identify new risk events
Risk Tracking
Watch-listed Risks
Assess Risk Criticality
Decision-analytic rules applied to rank-order identified risk events from "most to least" critical
4. Risk Mitigation Planning, Implementation, and Progress Monitoring
Risk Mitigation
3. Risk Prioritization Analysis
Risk events assessed as medium or high criticality might go into risk mitigation planning and implementation; low critical risks might be tracked/monitored on a watch list.

**Risk Management Process**

Very simply, a risk is a potential problem. It's an activity or event that may compromise the success of a software development project. Risk is the possibility of suffering loss, and total risk exposure to a specific project will account for both the probability and the size of the potential loss. Guesswork and crisis-management are never effective. Identifying and aggregating risks is the only predictive method for capturing the probability that a software development project will experience unplanned or inadmissible events. These include terminations, discontinuities, schedule delays, cost underestimation, and overrun of project resources. Risk management means risk containment and mitigation. First, you've got to identify and plan. Then be ready to act when a risk arises, drawing upon the experience and knowledge of the entire team to minimize the impact to the project.

Risk management includes the following tasks:

- **Identify** risks and their triggers
- **Classify** and prioritize all risks
- Craft a **plan** that links each risk to a mitigation
- **Monitor** for risk triggers during the project
- Implement the **mitigating action** if any risk materializes
- **Communicate** risk status throughout project

## 12.3.1 Identify and Classify Risks

Most software engineering projects are inherently risky because of the variety potential problems that might arise. Experience from other software engineering projects can help managers classify risk. The importance here is not the elegance or range of classification, but rather to precisely identify and describe all of the real threats to project success. A simple but effective classification scheme is to arrange risks according to the areas of impact.

**Five Types of Risk in Software Project Management**

For most software development projects, we can define five main risk impact areas:

- New, unproven technologies
- User and functional requirements
- Application and system architecture

- Performance
- Organizational

**New, unproven technologies:** The majority of software projects entail the use of new technologies. Ever-changing tools, techniques, protocols, standards, and development systems increase the probability that technology risks will arise in virtually any substantial software engineering effort. Training and knowledge are of critical importance, and the improper use of new technology most often leads directly to project failure.

**User and functional requirements:** Software requirements capture all user needs with respect to the software system features, functions, and quality of service. Too often, the process of requirements definition is lengthy, tedious, and complex. Moreover, requirements usually change with discovery, prototyping, and integration activities. Change in elemental requirements will likely propagate throughout the entire project, and modifications to user requirements might not translate to functional requirements. These disruptions often lead to one or more critical failures of a poorly-planned software development project.

**Application and system architecture:** Taking the wrong direction with a platform, component, or architecture can have disastrous consequences. As with the technological risks, it is vital that the team includes experts who understand the architecture and have the capability to make sound design choices.

**Performance:** It's important to ensure that any risk management plan encompasses user and partner expectations on performance. Consideration must be given to benchmarks and threshold testing throughout the project to ensure that the work products are moving in the right direction.

**Organizational:** Organizational problems may have adverse effects on project outcomes. Project management must plan for efficient execution of the project, and find a balance between the needs of the development team and the expectations of the customers. Of course, adequate staffing includes choosing team members with skill sets that are a good match with the project.

### 12.3.2 Risk Management Plan

After cataloging all of the risks according to type, the software development project manager should craft a risk management plan. As part of a larger, comprehensive project plan, the risk management plan outlines the response that will be taken for each risk—if it materializes.

### 12.3.3 Monitor and Mitigate

To be effective, software risk monitoring has to be integral with most project activities. Essentially, this means frequent checking during project meetings and critical events.

Monitoring includes:

- Publish project status reports and include risk management issues
- Revise risk plans according to any major changes in project schedule
- Review and reprioritize risks, eliminating those with lowest probability
- Brainstorm on potentially new risks after changes to project schedule or scope

When a risk occurs, the corresponding mitigation response should be taken from the risk management plan.

Mitigating options include:

- **Accept:** Acknowledge that a risk is impacting the project. Make an explicit decision to accept the risk without any changes to the project. Project management approval is mandatory here.
- **Avoid:** Adjust project scope, schedule, or constraints to minimize the effects of the risk.
- **Control:** Take action to minimize the impact or reduce the intensification of the risk.
- **Transfer:** Implement an organizational shift in accountability, responsibility, or authority to other stakeholders that will accept the risk.
- **Continue Monitoring:** Often suitable for low-impact risks, monitor the project environment for potentially increasing impact of the risk.

### 12.3.4 Communicate

Throughout the project, it’s vital to ensure effective communication among all stakeholders, managers, developers especially marketing and customer representatives. Sharing information and getting feedback about risks will greatly increase the probability of project success.

## 12.4 Risk Identification

Ensuring that adequate and timely risk identification is performed is the responsibility of the owner, as the owner is the first participant in the project. The sooner risks are identified, the sooner plans can be made to mitigate or manage them. Assigning the risk identification process to a contractor or an individual member of the project staff is rarely successful and may be considered a way to achieve the appearance of risk identification without actually doing it. It is important, however, that all project management personnel receive specific training in risk management methodology. This training should cover not only risk analysis techniques but also the managerial skills needed to interpret risk assessments. Because the owner may lack the specific expertise and experience to identify all the risks of a project without assistance, it is the responsibility of project directors to ensure that all significant risks are identified by the integrated project team (IPT). The actual identification of risks may be carried out by the owner's representatives, by contractors, and by internal and external consultants or advisors. The risk identification function should not be left to chance but should be explicitly covered in a number of project documents:

- Statement of work (SOW),
- Work breakdown structure (WBS),
- Budget,
- Schedule,
- Acquisition plan, and
- Execution plan.

**Methods Of Risk Identification**

There are a number of methods in use for risk identification. Comprehensive databases of the events on past projects are very helpful; however, this knowledge frequently lies buried in people's minds, and access to it involves brainstorming sessions by the project team or a significant subset of it. In addition to technical expertise and experience, personal contacts and group dynamics are keys to successful risk identification. Project team participation and face-to-face interaction are needed to encourage open communication and trust, which are essential to effective risk identification; without them, team members will be reluctant to raise their risk concerns in an open forum. While smaller, specialized groups can perform risk assessment and risk analysis, effective, ongoing risk identification requires input from the entire project team and from others outside it. Risk identification is one reason for early activation of the IPT is essential to project success.

The risk identification process on a project is typically one of brainstorming, and the usual rules of brainstorming apply:

- The full project team should be actively involved.
- Potential risks should be identified by all members of the project team.
- No criticism of any suggestion is permitted.
- Any potential risk identified by anyone should be recorded, regardless of whether other members of the group consider it to be significant.
- All potential risks identified by brainstorming should be documented and followed up by the IPT.

The objective of risk identification is to identify all possible risks, not to eliminate risks from consideration or to develop solutions for mitigating risks—those functions are carried out during the risk assessment and risk mitigation steps. Some of the documentation and materials that should be used in risk identification as they become available include these:

- Sponsor mission, objectives, and strategy; and project goals to achieve this strategy,
- Statement of Work (SOW),
- Project justification and cost-effectiveness (project benefits, present worth, rate of return, etc.),
- Work Breakdown Structure(WBS),
- Project performance specifications and technical specifications,
- Project schedule and milestones,
- Project financing plan,
- Project procurement plan,

- Project execution plan,
- Project benefits projection,
- Project cost estimate,
- Project environmental impact statement,
- Regulations and congressional reports that may affect the project,
- News articles about how the project is viewed by regulators, politicians, and the public, and
- Historical safety performance.

The risk identification process needs to be repeated as these sources of information change and new information becomes available.

There are many ways to approach risk identification. Two possible approaches are (1) to identify the root causes of risks—that is, identify the undesirable events or things that can go wrong and then identify the potential impacts on the project of each such event—and (2) to identify all the essential functions that the project must perform or goals that it must reach to be considered successful and then identify all the possible modes by which these functions might fail to perform. Both approaches can work, but the project team may find it easier to identify all the factors that are critical to success, and then work backward to identify the things that can go wrong with each one.

Risk identification should be performed early in the project (starting with pre project planning, even before the preliminary concept is approved) and should continue until the project is completed. Risk identification is not an exact science and therefore should be an ongoing process throughout the project, especially as it enters a new phase and as new personnel and contractors bring different experiences and viewpoints to risk identification. For this reason, the project director should ensure that the project risk management plan provides for periodic updates.

**Methods Of Qualitative Risk Assessment**

The goal of risk identification is not only to avoid omissions but also to avoid the opposite pitfall—of being distracted by factors that are not root causes but only symptoms. Treating the symptoms, rather than the root causes, will give the appearance of activity but will not solve the

Bottom of Form problem. Unfortunately, identification of symptoms is far easier than identification of root causes. Project owners should ensure that the risk identification process goes beyond the symptoms. While outside, disinterested reviewers can sometimes help perform this function, the following sections describe methods that can be used by project personnel to identify risks and their causes.

## Risk Screening

Following the initial risk identification phase, the project director should have a working list of risks that have been identified as potentially affecting the project. From this list, the project director should differentiate those that seem minor and do not require further attention from those that require follow-up, qualitative analysis, quantitative analysis, and active mitigation and management. This process requires some qualitative assessment of the magnitude and seriousness of each identified risk. Various methods that have been developed to assess failures in physical equipment and systems have also been applied in one form or another to project risks.

The commonly used risk tool shown in table below is a two by two matrix that allows assigning a risk to one of four quadrants based on a qualitative assessment of its relative impact (high or low) and the likelihood of its occurrence (high or low). Risks in the upper right quadrant  need the most attention. Finer gradations of impact and likelihood—for example, very high, high, medium, low, and very low (a five by five matrix)—would allow a more nuanced consideration of the attention needed.

**Risk Screening Based on Impact and Probability**

| | | | |
|---|---|---|---|
| Likelihood of Occurrence | High | | |
| | Low | | |
| | | Low | High |
| | | Relative Impact | |

**Low Impact, Low Probability**

Risks that can be characterized as both low impact and low likelihood of occurrence are essentially negligible and can usually be eliminated from active consideration. The main concern of the owner's project director is to monitor these factors sufficiently to determine that the impact or likelihood does not increase.

**High Impact, High Probability**

Risks that are characterized as both high impact and high likelihood of occurrence often cause a project to be terminated, or to fail if it is continued in spite of the risks. In this situation, the owner's management must determine if the project should be terminated or if the project is so mission critical or the potential benefits are so great that taking the risks is justified. Risk management does not imply that no risks are taken; it means that the risks taken should be calculated risks. For example, an owner may decide to proceed if there is a reasonable expectation that enough engineering or management effort can reduce either the impact or the likelihood of the events, such that the risk can become either low impact, high probability or low probability, high impact. Often such a decision is contingent on achieving the necessary risk reductions by some deadline.

**Low Impact, High Probability**

Low-impact, high-probability risks are those largely due to uncertainties about a number of elements that may be individually minor risks but that in the aggregate could amount to a significant risk. These include uncertainties concerning the actual costs of labor and materials (such as steel), the actual durations of activities, deliveries of equipment, productivity of the workforce, changes due to design development or the owner's preferences, and other uncertainties that are typically considered to lie within the natural variability of project planning, design, construction, and start-up (they do not include catastrophic events or radical design changes). Each of these uncertainties, taken alone, would have little impact on the project. However, taken together, there is the possibility that many of the estimates of these factors would prove to be too optimistic, leading Bottom of Form to cumulative effects such as performance shortfalls, schedule overruns, and cost overruns. Methods for dealing with such risks include

- Provision for adequate contingencies (safety factors) for budget and schedule.

- Improvement in the work processes in order to reduce the uncertainties. Prefabrication of major components to avoid the uncertainties of construction at a job site is one example of changing the normal process to reduce risks (although in this example the change may also introduce new risks, such as transportation of the components to the job site; thus the resolution of one risk may give rise to another).

**High Impact, Low Probability**

By definition, high-impact, low-probability events are rare occurrences, and therefore it is very difficult to assign probabilities to them based on historical records. Data do not exist and so subjective estimates of probabilities are necessary. However, the objective is not the scientific determination of accurate probabilities of rare events but the determination of what management actions should be taken to monitor, mitigate, and manage the risks. For example, if a certain risk is identified and management determines that some specific mitigation actions should be taken if the risk has a likelihood of more than 1 in 100 of occurring, then a precise characterization of the probability is unnecessary; the only issue is whether it is assessed to be more than 1 in 100 or less than 1 in 100**.**

**Pareto Diagrams**

One of the important uses of a good risk analysis is to determine where to apply management resources and what to leave alone, as management resources are not unlimited. One approach is to break down the uncertainties into manageable parts. Pareto diagrams are one way to show the sources of uncertainty or impact in descending order. This form of presentation makes explicit those activities that have the greatest effect on the project completion date or cost and that therefore require the greatest management attention. The project director or manager must then determine whether the high-ranking events are (1) truly root causes or (2) simply work packages or activities that may reflect underlying causes but are themselves symptoms. The resulting analysis can provide guidance for managers to reduce, mitigate, buffer, or otherwise manage these sources of uncertainty.

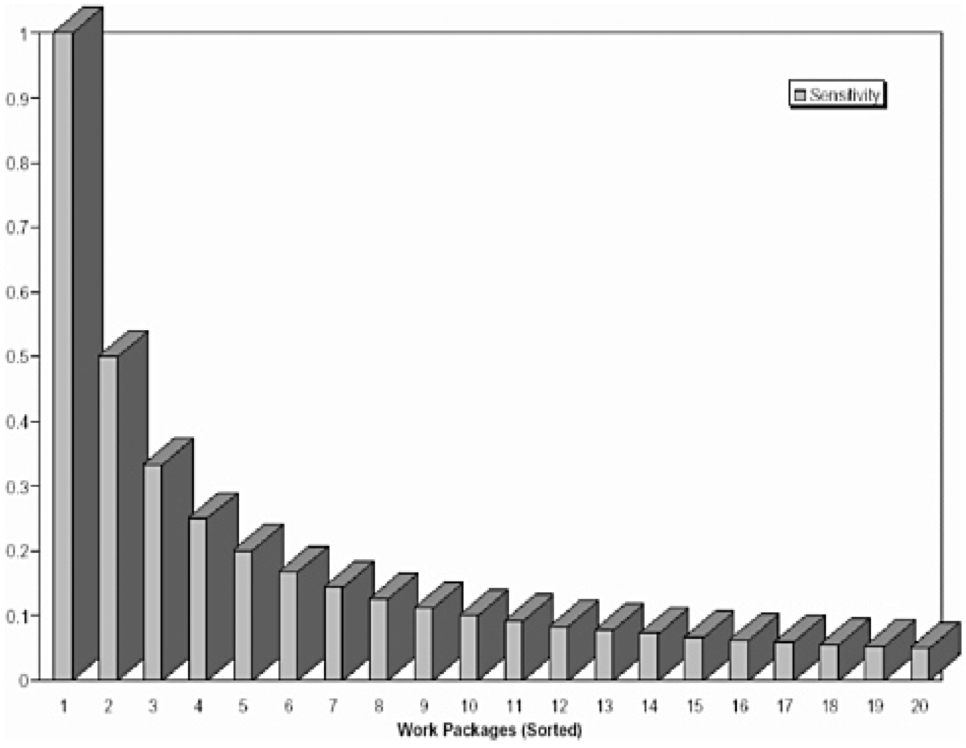
As a simple illustration, suppose we are interested in determining which work packages have the greatest effects on the uncertainty in the total cost. First, we estimate the uncertainty, or variance, in the cost of each individual work package. Second, we estimate the correlations or associations between each pair of work packages. Then, by elementary second-moment theory (Benjamin and Cornell, 1970), the sensitivity of the uncertainty in the total project cost with respect to each work package is proportional to the combination of the activity uncertainties and the correlations between activities. That is, the uncertainty in the total cost is affected not only by the uncertainty in each work package but also by how much each work package affects, and is affected by, the others.

As an elementary example, the uncertainty in the cost of a construction project may be more sensitive to outdoor activities than to indoor activities because unusually bad weather can cause a number of outdoor activities to run over budget and over schedule simultaneously, whereas indoor activities are typically not linked so tightly to the weather. By tabulating these values for all work packages, and sorting them from largest to smallest, we can identify those work packages with the largest sensitivities, which are those to which the project manager should give the highest priority. If we do this for a project of, say, 20 work packages and sort them according to the largest values of the sensitivities, we can then plot a Pareto diagram. The absolute values of the sensitivities have no importance; the only concern is the relative values.

## Failure Modes and Effects Analysis

In project risk assessment, a failure can be any significant event that the sponsor does not want to happen—a budget overrun, a schedule overrun, or a failure to meet scope, quality, or mission performance objectives. While risks may arise from specific causes, they may also be the result of general environmental conditions that are not limited to specific times and places but are pervasive throughout the project. The objective of failure modes and effects analysis is the identification of root or common causes, which may affect the project as a whole. Often this identification is facilitated by methodically considering the project function by function,

[1] All probability distributions may be characterized by their moments. Second-moment theory is the use of the second moments of probability distributions—that is, means, variances, and covariances (or correlation coefficients), instead of full probability distribution functions. As probability distributions are subjective and therefore not capable of precise definition, this approximate method can greatly simplify many calculations and, more importantly, provide the risk analyst with insight into the effects of uncertainty on project outcomes.



**Peroto Diagram**

Identification of potential risks that turn out, upon further assessment, to be negligible is a waste of time; however, failure to identify potential risks that turn out to be serious is a threat to the project. Therefore, the project director should err on the side of caution when identifying possible risks.

Failure modes and effects analysis (FMEA) is a discipline or methodology to assist in identifying and assessing risks qualitatively. It is a method for ranking risks for further investigation; however, it is not a method for quantifying risks on a probabilistic basis (Breyfogle, 1999). FMEA is typically based on a subjective assessment of the relative magnitudes of the impacts of the risk events on the project (often on a scale from 1 to 10), multiplied by the relative likelihood that the risk event will occur (also on a scale from 1 to 10). In addition, a third parameter may be included to assess the degree of warning that the project will have regarding the actual occurrence of the risk event (again on a scale from 1 to 10). This third parameter may give some management support by establishing early warning indicators for specific serious

The purpose of assigning these values for all significant risks is only to rank the risks and to set priorities for subsequent quantitative analysis of the significant risks. In the absence of more quantitative factors, such as sensitivity analysis, the failure modes, or better, all root causes, can be used to rank the risks. One can prepare a Pareto chart that shows the risks ordered by possible impact or by the combination of impact and likelihood of occurrence. Then risk mitigation efforts can first address the failure mode or root cause with the highest impact and work from there.

The three factors—severity, likelihood, and leading indicators—interact. For example, if the project is the construction of a facility in a flood plain or an area with poor drainage, then a failure mode could be flooding of the work site. Project management cannot affect the frequency of floods, so risk management must focus on trying to reduce the severity of the impact of a flood. If the control method is to buy flood insurance and then evacuate personnel and abandon the site if the water rises, then measuring the height of the water (the "Nilometer" method) may be a sufficient indicator. If the control method is to reduce the severity of loss by placing sandbags around the perimeter and renting pumps, then measuring the water height may have little impact on the mitigation effort; but measuring the rainfall across the watershed may be more appropriate because it allows time to implement the control. If the control method is to build a cofferdam around the site before constructing anything else, then the choice of leading indicator may be irrelevant.

Efforts to mitigate the risks will focus on the impact, likelihood, and detect ability of the most serious risk or its root causes and will try to reduce these factors until this risk becomes as low as or lower than the next higher risk. As this process continues, the most important risks will be reduced until there are a number of risks essentially the same and a number of other risks all lower than the first group. The first group will require specific management actions and may require constant monitoring and attention throughout the project. The second group will be monitored, but with lower priority or frequency. The first group is considered the critical group, much like the critical-path activities in a network schedule; the second group is the noncritical group, which must be watched primarily to see that none of the risks from this group become critical.

It should be emphasized that this form of risk assessment is qualitative and relative, not quantitative and absolute. It is primarily for distinguishing between risks that require follow-up and management, because of high impact or high likelihood (or both), and risks that do not appear to require follow-up, because of both low impact and low likelihood. It should be clearly understood that there is no quantitative assessment of the overall risk to the total project: The severity factors are not estimated in terms of loss of dollars, the likelihoods of occurrence are not probabilities, and there is no cost-benefit analysis of the risks versus the control methods. The analysis only identifies risk priorities in a methodical way to help direct further risk management activities. It is left to the judgment of the project engineers, designers, and managers to determine the appropriate risk mitigation and control measures to achieve an acceptable level of risk. Note especially that risks with a low likelihood of occurrence but very high severities may require follow-up and management action.

Due to changes in project conditions or perceptions, even risks that appear to have low impact and high likelihood at one time may appear differently at another. Therefore, the owner's representatives have the responsibility to reevaluate all failure modes and effects periodically to ensure that a risk previously considered negligible has not increased in either impact or likelihood to a level requiring management attention.

## Project Definition Rating Index

The Project Definition Rating Index (PDRI) is an example of an additive qualitative risk assessment tool (CII, 1996, 1999). The PDRI is used in front-end project planning to help the project team assess project scope definition, identify risk elements, and subsequently develop mitigation plans. It includes detailed descriptions of issues and a weighted checklist of project scope definition elements to jog the memory of project team participants. It provides the means to assess risk at various stages during the front-end project planning process and to focus efforts on high-risk areas that need additional definition. The PDRI facilitates the project team's assessment of risks in the project scope, cost, and schedule. Each risk element in the PDRI has a series of five predetermined weights. Once the weights for each element are determined they are added to obtain a score for the entire project. This score is statistically correlated with project performance to estimate the level of certainty in the project baseline.

## Methods Of Quantitative Risk Analysis

After risk factors are assessed qualitatively, it is desirable to quantify those determined by screening activities to be the most significant. It cannot be repeated too often that the purpose of risk assessment is to be better able to mitigate and manage the project risks—not just to compute project risk values. The assessment of risks attributed to elements completely out of project management control—such as force majeure, acts of God, political instability, or actions of competitors—may be necessary to reach an understanding of total project risk, but the risk assessment should Bottom of Form be viewed as a step toward identifying active measures to manage all risks, even those considered outside the control of project managers, not to support a passive attitude toward risks as inevitable.

It is often desirable to combine the various identified and characterized risk elements into a single quantitative project risk estimate. Owners may also be interested in knowing the total risk level of their projects, in order to compare different projects and to determine the risks in their project portfolios. This estimate of overall project risk may be used as input for a decision about whether or not to execute a project, as a rational basis for setting a contingency, and to set priorities for risk mitigation.

While probabilistic risk assessment methods are certainly useful in determining contingency amounts to cover various process uncertainties, simple computation methods are often as good as, or even better than, complex methods for the applications discussed here. Owner's representatives should be proficient in simple statistical approaches for computing risk probabilities, in order to be able to check the numbers given to them by consultants and contractors. When addressing probabilistic risk assessment, project directors should keep in mind that the objective is to mitigate and manage project risks and that quantitative risk assessment is only a part of the process to help achieve that objective.

There are many available methods and tools for quantitatively combining and assessing risks. Some of the most frequently used methods are discussed briefly below.

**Multivariate Statistical Models**

Multivariate statistical models for project costs or durations are derived from historical data. The models are typically either top-down or parametric and do not contain enough detail to validate bottom-up engineering estimates or project networks.

These methods are objective in that they do not rely on subjective probability distributions elicited from (possibly biased) project advocates. Analysts build linear or nonlinear statistical models based on data from multiple past projects and then compare the project in question to the models. The use of such statistical models is desirable as an independent benchmark for evaluating cost, schedule, and other factors for a specific project, but statistically based methods require a large database of projects, and many owners do not perform enough projects or expend the effort to create such databases. Owners who have performed many projects but have not developed usable historical project databases have an opportu nity to improve their competence in project and program management by organizing their own data. Computational methods such as resampling and bootstrapping are also used when data are insufficient for direct statistical methods. The bootstrap method is a widely used computer-based statistical process originally developed by Efron and Tibshirani (1993) to create a proxy universe through replications of sampling with replacement of the original sample. Bootstrapping is used to estimate confidence levels from limited samples but is not applicable for developing point estimates.

## Event Trees

Event trees, also known as fault trees or probability trees, are commonly used in reliability studies, probabilistic risk assessments (for example, for nuclear power plants and NASA space probes), and failure modes and effects analyses. The results of the evaluations are the probabilities of various outcomes from given faults or failures. Each event tree shows a particular event at the top and the conditions causing that event, leading to the determination of the likelihood of these events. These methods can be adapted to project cost, schedule, and performance risk assessments.

## System Dynamics Models

Projects with tightly coupled activities are not well described by conventional project network models (which prohibit iteration and feedback). Efforts to apply conventional methods to these projects can lead to incorrect conclusions, counterproductive decisions, and project failures. In contrast, system dynamics models (Forrester, 1969) describe and explain how project behavior and performance are driven by the feedback loops, delays, and nonlinear relationships in processes, resources, and management. System dynamics models can be used to clarify and test project participants' assumptions as well as to design and test proposed project improvements and managerial policies. Because system dynamics models are based on dynamic feedback the models can also be used to evaluate the impacts of various failure modes or root causes, particularly in cases where the root causes can be identified but the ripple effect of their impacts is difficult to estimate with any confidence.

System dynamics models have been effectively used for project evaluation, planning, and risk assessment (Cooper, 1980; Lyneis, Cooper, and Els, 2001; Ford and Sterman, 2003). Although the use of these models is not standard practice for project planning and risk management, they can significantly help owners to improve their understanding of project risks.

## Sensitivity Analysis

Sensitivity analysis of the results of any quantitative risk analysis is highly desirable. A sensitivity coefficient is a derivative: the change in some outcome with respect to a change in some input. Even if the probability of a particular risk cannot be determined precisely, sensitivity analysis can be used to determine which variables have the greatest influence on the risk. Because a primary function of risk analysis is to break down the problem into essential elements that can be addressed by management, sensitivity analysis can be very useful in determining what decisions the manager should make to get the desired results—or to avoid undesired results. In the absence of hard data, sensitivity analysis can be very useful in assessing the validity of risk models.

## Project Simulations

Project simulations are group enactments or simulations of operations, in which managers and other project participants perform the project activities in a virtual environment before undertaking them on the project. This type of simulation may or may not be supported by computers; the emphasis is not on the computer models but rather on the interactions of the participants and the effects of these interactions on project outcomes. For this reason, project simulations are very good for team building before a project actually starts up. They are not inexpensive, but the cost is generally comparable to the costs of the other techniques cited here, and they can be very cost-effective in the long run, compared to the typical approach of jumping into major projects with little or no preparation of the personnel and their working relationships. Engineering and construction contractors have developed project simulation methods (Halpin and Martinez, 1999), and owners can develop their own or specify that their contractors should perform such simulations before a project starts, in conjunction with the other preproject planning efforts**.**

## Stochastic Simulation Models

Stochastic simulation models are computerized probabilistic simulations that, for computational solution, typically use random number generators to draw varieties from probability distributions. Because the computer simulation is performed with random numbers, these methods are also called Monte Carlo simulations. The objective of the simulation is to find the uncertainties (empirical probability distributions) of some dependent variables based on the assumed uncertainties (subjective probability distributions) of a set of independent variables, when the relation ships between the dependent and independent variables are too complex for an analytical solution. Thus each iteration (random simulation) may be considered an experiment, and a large number of these experiments give insights into the probabilities of various outcomes. Monte Carlo simulation is typically used to combine the risks from multiple risk factors and as such is useful to determine whether the total risk of a project is too great to allow it to proceed or to determine the appropriate amount of contingency.

Stochastic simulations differ from multivariate statistical models because they are typically not based on hard data. They can be useful in the absence of real data in that they are based on subjective assessments of the probability distributions that do not require large databases of previous project information. An often-cited weakness of this method is that subjective assessments of probability distributions often lack credibility, because they may be influenced by bias. This can be overcome to some degree by a carefully structured application of expert judgment (Keemey and von Winterfeldt, 1991).

As is the case with all the other computer methods for quantitative risk analysis discussed here, the validity of the method lies entirely in the validity of the probabilistic models. Monte Carlo simulation is very versatile because it can be applied to virtually any probabilistic model. However, the validity of the results may sometimes be suspect, due to the following factors:

- The independent variables may not actually be independent;
- The number of iterations in the simulation may be insufficient to produce statistically valid results; or
- The probability distributions assumed for the independent variables are subjective and may be biased if they are provided by project proponents.

It is certainly possible to develop project-specific cost models, for example, by using causal parameters that are totally independent. However, many risk analyses are not based on project-specific models but simply adopt the standard engineering additive cost models, in which the total cost is the sum of work package costs. The simulations simply add up the uncertainties associated with work packages, but they may be inaccurate because these work packages are not necessarily independent. It is computationally much easier to perform Monte Carlo simulation if the analyst avoids the need to consider interactions between variables by simply assuming that all variables are independent; however, an analysis without consideration of common mode failure can lead to an under estimation of total project risk. In project risk assessment, a common mode could be an event or environmental condition that would cause many cost variables to tend to increase (or decrease) simultaneously. It is widely recognized that a single event can cause effects on a number of systems (i.e., the ripple effect). If the event occurs, the costs of these systems will all increase, whereas if it does not occur, they will remain within the budget. Thus these affected costs are definitely not statistically independent.

Collaboration between people who are very conversant with the specific risks of the project and those who are familiar with probabilistic methods is typically required to reduce bias and to produce realistic quantification of project risks. Project owners should ensure that the probabilistic inputs are as objective and unbiased as possible and that the reasons for choosing specific probability distributions are adequately documented.

As with any method, the use of stochastic simulation requires quality control. The owner's policies and procedures on Monte Carlo simulation should include cautions to project directors and managers about the limitations of this method as it is commonly applied. The project director is generally not a specialist in Monte Carlo simulation, and does not need to be, but should understand the advantages and limitations of this approach. This is particularly true now that Monte Carlo simulation is readily available through common spreadsheet software and so can be used by people with little knowledge of statistics. A project director should know enough to be able to critically evaluate the stochastic simulation results for plausibility and should not accept the results just because they come from a computer.

It is common for Monte Carlo simulations to use far fewer iterations than the minimum normally required to get statistically valid answers. But simulations with insufficient iterations may underestimate the probability in the tails of the distributions, which is where the risks are. Therefore, a simulation with fewer random samples may indicate more or less risk than one with more iteration. There are mathematical formulas (Breyfogle, 1999) that can be used to compute the minimum number of iterations for acceptable confidence limits on the means or the values in the tails of the distribution. If a consultant or contractor is performing Monte Carlo simulations for risk assessments, it would be prudent for the owner's project director to review the confidence limits on all values computed using Monte Carlo simulation, to ensure that a sufficient number of iterations has been performed.

The use of Monte Carlo and other techniques for mathematically combining the risks of individual work packages into a single project risk number should not obscure the fact that the objective is to manage the risks. As typically used, Monte Carlo simulations tend to be focused on total risk probabilities, not on sensitivity analysis, risk prioritization, or assessing possible outcomes from different proposed risk management policies.

## Additive Models

Additive models, as the name implies, are those in which the combination of risk factors is based on simple addition. An example is the summation of cost elements to generate the total project cost, or the summation of activity durations to generate the total project duration. These models are relatively simple programs based on the summation of moments, derived from probability theory, to combine risks for dependent as well as independent variables. If the objective is simply to find the probability distribution of the project cost estimate as the sum of a number of work packages or activities, stochastic simulation is unnecessary. One advantage of simple additive models is that they are easily understood, and it is usually obvious which activities contribute the most to the total project uncertainty and which do not. This method is the basis for the program evaluation and review technique (PERT) for determining uncertainty in project completion times.

In bottom-up project cost estimating, the total cost is simply the sum of the costs in the WBS work packages. This is a purely linear relationship. Therefore, estimating the uncertainty in the total cost requires only summing the uncertainties in the individual cost accounts, modified by the dependencies between them. Probability theory tells us that we can compute the moments of the probability distribution of the total project cost by summing the moments of the uncertainties in all the individual cost accounts (Burlington and May, 1953; Hald, 1952). The number of moments can be approximated to some finite number. (This is a very common method of approximation in engineering—for example, the truncation of a Taylor Series after one term in order to gain a linear equation.) The second-moment approach (Benjamin and Cornell, 1970) uses the first two moments, i.e., the mean and the variance, and neglects the third (skewness) and higher. The second-moment approach does not deal with full probability distributions but uses only the means, variances, and covariances (the first two moments) to characterize uncertainties.

This approximation is justified because it is very difficult or even impossible to estimate higher moments (skewness, kurtosis, etc.) with any accuracy, even if one were in possession of large historical databases. In most cases of risk assessment, the probability distributions are largely subjective and based on judgment and experience rather than hard data. There is little point in making highly precise computer calculations on numbers that cannot be estimated accurately anyway.

There are some additional advantages of the second-moment approach:

- Priorities for risk mitigation can be obtained from a Pareto analysis using just the uncertainty in each individual risk factor and the correlations between risk factors.
- Sensitivity analyses are easily performed.
- As a project progresses, the estimates of the uncertainties in future cost accounts or activities can readily be revised, based on the past performance of the project itself. This is one of this method's most useful properties. By comparing the actual performance on completed work packages, activities, or milestones with the prior estimated uncertainties, one obtains revised estimates of the work packages, activities, or milestones yet to come.

Through second-moment analysis, project directors can use the information and experience on the actual project to revise the estimates of the work to go. This approach can be a valuable tool for program managers, if each project director is required to report the updated, revised cost at completion, including the confidence bounds on this estimate, for every reporting period. Because this method looks forward instead of backward, as most other project management methods do (including earned value analysis), unfavorable revisions to either the expected cost at completion or the uncertainty in the cost at completion should trigger management action. Conversely, favorable revisions to either the expected cost at completion or the uncertainty in the cost could allow management reserves to be reallocated to other projects with greater needs.

The second-moment method provides a simple, convenient method for the adjustment of risks, and hence the adjustment of the required contingencies, as a project proceeds and data are obtained on how well or badly it is performing. The objective of this approach is to react as soon as possible to information on recent project performance that tends to confirm or to refute the current estimates. The key control parameter is the estimated cost (or time) at completion. For example, if the best estimate of the cost at completion, updated with the most recent progress information, is higher than the original estimate, then, assuming no scope changes, either the risk of overrunning the budget is greater than originally estimated, or program management corrective action may be needed to bring the project back on target. Conversely, if the updated best estimate of the cost at completion is the same as or lower than the original estimate, then the required contingency can be decreased. In this approach, the estimates of all future work packages are updated as the actual costs for each completed work package become available through the cost reporting system.

**Summary of Risk Analysis Tools**

| Tool | Characteristics |
|---|---|
| Two-dimensional impact/ probability | Qualitative, simple to use and most frequently used, can be expanded to three or more dimensions, and can be combined with FMEA |
| Pareto diagram | Simple qualitative method for prioritizing risk elements |
| Failure modes and effects analysis (FMEA) | Qualitative, used for initial screening only, effective in a team environment |
| Project Definition Rating Index | Qualitative, used in front-end project planning, effective in a team environment |
| Multivariate statistical model | Quantitative, requires historical database |
| Event tree | Quantitative, rarely used for risk analysis |
| System dynamics model | Both qualitative and quantitative, rarely used but effective, requires skilled modelers |
| Sensitivity analysis | Quantitative, useful regardless of which other process used, useful in absence of hard data |
| Project simulation | Both qualitative and quantitative, useful for team building, expensive to implement |
| Stochastic simulation | Quantitative, frequently used, often misused, so limitations must be made clear |
| Additive model | Quantitative, can be adjusted as project progresses |

## 12.5 Risk Projection

Attempts to rate each risk in two ways

- The probability that the risk is real
- The consequences of the problems associated with the risk, should it occur.

Project planner, along with other managers and technical staff, performs four risk projection activities:

(1) Establishes a measure that reflects the perceived likelihood of a risk

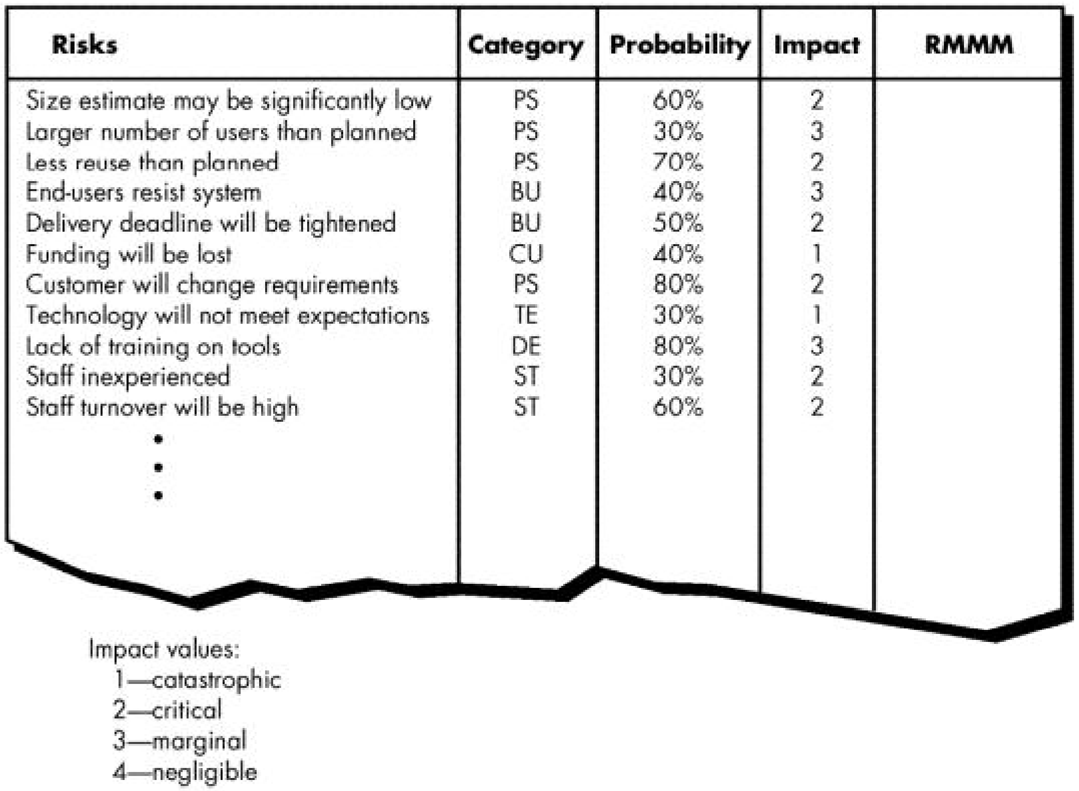
(2) Delineate the consequences of the risk

(3) Estimate the impact of the risk on the project and the product

(4) Note the overall accuracy of the risk projection so that there will be no misunderstandings.

**Developing a Risk Table**

Risk table provides a project manager with a simple technique for risk projection



| Risks | Category | Probability | Impact | RMMM |
|---|---|---|---|---|
| Size estimate may be significantly low | PS | 60% | 2 | |
| Larger number of users than planned | PS | 30% | 3 | |
| Less reuse than planned | PS | 70% | 2 | |
| End-users resist system | BU | 40% | 3 | |
| Delivery deadline will be tightened | BU | 50% | 2 | |
| Funding will be lost | CU | 40% | 1 | |
| Customer will change requirements | PS | 80% | 2 | |
| Technology will not meet expectations | TE | 30% | 1 | |
| Lack of training on tools | DE | 80% | 3 | |
| Staff inexperienced | ST | 30% | 2 | |
| Staff turnover will be high | ST | 60% | 2 | |
| • | | | | |
| • | | | | |
| • | | | | |

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

**Risk Table**

**Steps in Setting up Risk Table**

(1) Project team begins by listing all risks in the first column of the table. It is accomplished with the help of the risk item checklists.

(2) Each risk is categorized in the second column

(e.g., PS implies a project size risk, BU implies a business risk).

(3) The probability of occurrence of each risk is entered in the next column of the table.

The probability value for each risk can be estimated by team members individually.

(4) Individual team members are polled in round-robin fashion until their assessment of risk probability begins to converge.

**Assessing Impact of Each Risk**

(1) Each risk component is assessed using the Risk Charcterization Table and impact category is determined.

(2) Categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.

1. A risk that is 100 percent probable is a constraint on the software project.
2. The risk table should be implemented as a spreadsheet model. This enables easy manipulation and sorting of the entries.
3. A weighted average can be used if one risk component has more significance for the project.

(3) Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact.

- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom.

(4) Project manager studies the resultant sorted table and defines a cutoff line.

· Cutoff line (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention.

· Risks below the line are re-evaluated to accomplish second-order prioritization.

· Risk impact and probability have a distinct influence on management concern.

- Risk factor with a high impact but a very low probability of occurrence should not absorb a significant amount of management time.
- High-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.
- All risks that lie above the cutoff line must be managed.
- The column labeled RMMM contains a pointer into a Risk Mitigation, Monitoring and Management Plan

**Assessing Risk Impact**

· Three factors determine the consequences if a risk occurs:

✓ **Nature** of the risk - the problems that are likely if it occurs.

e.g., a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project.

✓ **Scope** of a risk - combines the severity with its overall distribution (how much of the project will be affected or how many customers are harmed?).

✓ **Timing** of a risk - when and how long the impact will be felt.

· Steps recommended determining the overall consequences of a risk:

1. Determine the average probability of occurrence value for each risk component.
2. Using Figure 1, determine the impact for each component based on the criteria shown.
3. Complete the risk table and analyze the results as described in the preceding sections.

Overall **risk exposure**, **RE**, determined using:

**RE = P x C**

**P** is the probability of occurrence for a risk

**C** is the the cost to the project should the risk occur.

### Check Your Progress

1. ______________ means risk containment and mitigation.

2. I am a discipline or methodology used to assist in identifying and assessing risks qualitatively. Who am I?

3. Expand PS.

4. List out the factors that determine consequence if risk occurs.

5. Match the following

| | |
|---|---|
| a. Pareto Diagram | i. Quantitative Method that requires historical database |
| b. Additive Model | ii. Qualitative Method Effective for front end project planning in team environment. |
| c. Multivariate Statistical Model | iii. Qualitative Method for Prioritizing Risk Elements |
| d. Project Definition Rating Index | iv. Quantitative Method that adjust as project progresses |

## 12.6 Risk Refinement

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

One way to do this is to represent the risk in condition-transition-consequence (CTC) format. That is, the risk is stated in the following form: Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk, we can write as given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

**Subcondition 1.** Certain reusable components were developed by a third party with no knowledge of internal design standards.

**Subcondition 2.** The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

**Subcondition 3.** Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined subconditions remains the same (i.e., 30 percent of software components must be customer engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

## 12.7 Risk Mitigation, Monitoring and Management (RMMM) Plan

A risk management strategy can be defined as a software project plan or the risk management steps. It can be organized into a separate Risk Mitigation, Monitoring and Management Plan. The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

Teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet . In most cases, the RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives:

(1) To assess whether predicted risks occur.

(2) To ensure that risk aversion steps defined for the risk are being properly applied; and

(3) To collect information that can be used for future risk analysis.

**Effective strategy must consider three issues:**

- risk avoidance
- risk monitoring
- risk management and contingency planning. Proactive approach to risk – avoidance strategy. Develop risk mitigation plan. Develop a strategy to mitigate this risk for reducing turnover. Meet with current staff to determine causes for turnover. Mitigate those causes that are under our control before the project starts.
- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner. Project manager monitors for likelihood of risk, Project manager should monitor the effectiveness of risk mitigation steps. Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. RMMM steps incur additional project cost.

**The Rmmm Plan**

Risk Mitigation, Monitoring and Management Plan (RMMM) – documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. Risk monitoring is a project tracking activity

**Three primary objectives:**

- Assess whether predicted risks do, in fact, occur
- Ensure that risk aversion steps defined for the risk are being properly applied
- Collect information that can be used for future risk analysis.

## 12.8 Summary

Risk management is an extensive discipline, and we've only given an overview here. We leave you with a checklist of best practices for managing risk on your software development and software engineering projects:

- Always be forward-thinking about risk management. Otherwise, the project team will be driven from one crisis to the next.
- Use checklists, and compare with similar previous projects.
- Prioritize risks, ranking each according to the severity of exposure.
- Develop a top-10 or top-20 risk list for your project. Like most project managers, you can probably reuse this list on the next project!
- Vigorously watch for surfacing risks by meeting with key stakeholders—especially with the marketing team and the customer.
- As practicable, split larger risks into smaller, easily recognizable and readily-manageable risks.
- Strongly encourage stakeholders to think proactively and communicate about risks throughout the entire project

## 12.9 Check your Answers

1. Risk Management
2. Failure modes and effects analysis (FMEA)
3. Project Size Risk
4. Nature, Scope and Timing of Risk
5. a. iii b. iv c. i d. ii

## 12.10 Model Questions

1. Explain the methods of quantitative risk analysis.
2. Explain the Risk Management Task with a neat flow diagram.
3. What is RMMM Plan?
4. Explain the Risk Refinement in Risk management.
5. What is the use of Pareto Diagram?
6. Explain the methods of Quantitative Risk Analysis.
7. What is Stochastic Simulation Model?
8. Define Additive Model.
9. List out the activities of Risk Projection.
10. What is Cuttoff Line?

# LESSON-13

# SOFTWARE REVIEW TECHNIQUES

## 13.2 Learning Objective

Upon successful completion of this lesson, you will be able to:

- Describe the software review techniques.
- Understand what formal and informal review is and differentiate between them.
- Define what Software Quality Assurance is.
- Describe the tasks of Software Quality Assurance.

### Structure

## 13.2 Introduction

Software review is an important part of ”Software Development Life Cycle (SDLC)” that assists software engineers in validating the quality, functionality, and other vital features and components of the software. It is a complete process that involves testing the software product and ensuring that it meets the requirements stated by the client.

## 13.3 Software Review Techniques

### Software Review

It is systematic examination of a document by one or more individuals, who work together to find & resolve errors and defects in the software during the early stages of Software Development Life Cycle (SDLC). Usually performed manually, software review is used to verify various documents like requirements, system designs, codes, ”test plans”, & ”test cases”.

A review is a systematic examination of a document by one or more people with the main aim of finding and removing errors early in the software development life cycle. Reviews are used to verify documents such as requirements, system designs, code, test plans and test cases. Reviews are usually performed manually while static analysis of the tools is performed using tools.

### Importance of Review Process

- Productivity of Dev team is improved and timescales reduced because the correction of defects in early stages and work-products will help to ensure that those work-products are clear and unambiguous.
- Testing costs and time is reduced as there is enough time spent during the initial phase.
- Reduction in costs because fewer defects in the final software.

### Types of Defects during Review Process

- Deviations from standards either internally defined or defined by regulatory or a trade organisation.
- Requirements defects.

- Design defects.
- Incorrect interface specifications.

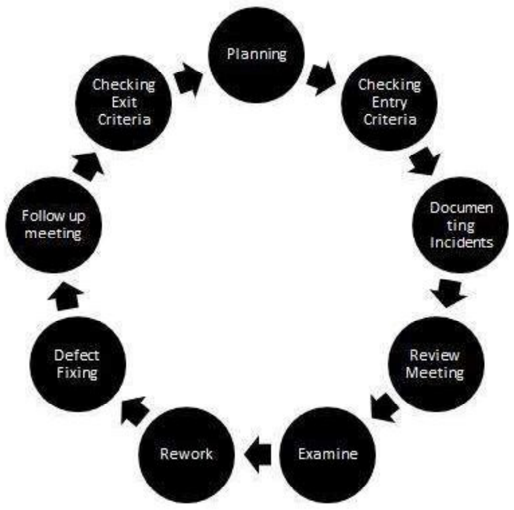


**Figure Workflow of Review Stages**

**Need for Software Review**

The reasons that make software review an important element of software development process are numerous. It is one such methodology that offers an opportunity to the development team & the client, to get clarity on the project as well as its requirements. With the assistance of software review, the team can verify whether the software is developed as per the requested requirements or not, and make the necessary changes before its release in the market. Other important reasons for Software Review are:

- It improves the productivity of the development team.
- Makes the process of testing time & cost effective, as more time is spent on testing the software during the initial development of the product.
- Fewer defects are found in the final software, which helps reduce the cost of the whole process.
- The reviews provided at this stage are found to be cost effective, as they are identified at the earlier stage, as the cost of rectifying a defect in the later stages would be much more than doing it in the initial stages.

- In this process of reviewing software, often we train technical authors for defect detection process as well as for ”defect prevention process”.
- It is only at this stage the inadequacies are eliminated.
- Elimination of defects or errors can benefit the software to a great extent. Frequent check of samples of work and identification of small time errors can lead to low error rate.
- As a matter of fact, this process results in dramatic reduction of time taken in producing a technically sound document.

## 13.4 Types of Software Reviews

There are mainly three types of software reviews, all of which are conducted by different members of the team who evaluate various aspects of the software. Hence, the types of software review are:

1. **Software Peer Review:** Peer review is the process of evaluating the technical content and quality of the product and it is usually conducted by the author of the work product, along with some other developers. According to ”Capacity Maturity Model”, the main purpose of peer review is to provide ”a disciplined engineering practice for detecting or correcting defects in the software artifacts, preventing their leakage into the field operations”. In short, peer review is performed in order to determine or resolve the defects in the software, whose quality is also checked by other members of the team.

   Types of Peer Review:

o **“Code Review”:** To fix mistakes and to remove vulnerabilities from the software product, systematic examination of the computer source code is conducted, which further improves the quality & security of the product.

o **“Pair Programming”:** This is a type of code review, where two programmers work on a single workstation and develop a code together.

o **Informal:** As suggested by its name, this is an informal type of review, which is extremely popular and is widely used by people all over the world. Informal review does not require any documentation, **”entry criteria”**, or a large group of people. It is a time saving process that is not documented.

- **“Walkthrough”**: Here, a designer or developer lead a team of software developers to go through a software product, where they ask question and make necessary comments about various defects & errors. This process differs from **”software inspection”** and technical review in various aspects.

- **Technical Review:** During the process of technical review a team of qualified personnels review the software and examine its suitability to define its intended use as well as to identify various discrepancies.

- **Inspection:** This is a formal type of peer review, wherein experienced & qualified individuals examine the software product for bugs and defects using a defined process. Inspection helps the author improve the quality of the software.

2. **Software Management Review:** These reviews take place in the later stages by the management representatives. The objective of this type of review is to evaluate the work status. Also, on the basis of such reviews decisions regarding downstream activities are taken.

3. **Software Audit Reviews: “Software Audit”** review or software review is a type of external review, wherein one or more auditors, who are not a part of the development team conduct an independent examination of the software product and its processes to assess their compliance with stated specifications, standards, and other important criterion’s. This is done by managerial level people.

## 13.4.1 Process of Software Review

The process of software review is a simple one and is common for all its types. It is usually implemented by following a set of activities, which are laid down by IEEE Standard 1028. All these steps are extremely important and need to be followed rigorously, as skipping even a single step can lead to a complication with the development process, which can further affect the quality of the end product.

1. **Entry Evaluation:** A standard check-list is used by entry criteria in order to ensure an ideal condition for a successful review.

2. **Management Preparation:** During this stage of the process, a responsible management ensures that the software review has all the required resources, which includes things like staff, time, materials, and tools.

3. **Review Planning:**To undergo a software review, an objective is identified. Based on the objective, a recognized team of resources is formed.

4. **Preparation:** The reviewers are held responsible for preparing group examination to do the reviewing task.

5. **Examination and Exit Evaluation:** In the end, the result made by each reviewer is combined all together. Before the review is finalized, verification of all activities is done that are considered necessary for an efficacious software review.

## 13.5 Informal Reviews

Unlike Formal Reviews, Informal reviews are applied multiple times during the early stages of software development process. The major difference between the **formal and informal reviews** is that the former follows a formal agenda, whereas the latter is conducted as per the need of the team and follows an informal agenda. Though time saving, this process is not documented and does not require any entry criteria or large group of members.

Features of Informal Review:

- Conducted by a group of 2-7 members, which includes the designer an any other interested party.
- Here the team identifies errors & issues as well as examine alternatives.
- It is a forum for learning.
- All the changes are made by the software designer.
- These changes are verified by other project controls.
- The role of informal review is to keep the author informed and to improve the quality of the product.

Informal review is the most common and widely use review technique. There are so many advantages of informal review over formal and well documented review but the key advantage is time saving.To conduct the informal review documentation, minimum number of members or entry criteria etc. are not mandatory so this can be consider as a better way of review.

## 13.5.1 Types of Informal Review

### Walk through

Method of conducting informal group/individual review is called walk through, in which a designer or programmer leads members of the development team, testing team and other interested parties through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems or may suggest improvement on the article, walk through can be pre-planned or can be conducted at need basis and generally people working on the work product are involved in the walk through process.

The Purpose of walk through is to:

- Find problems
- Discuss alternative solutions
- Focusing on demonstrating how work product meets all requirements.

### Peer Review

Peer review means that an action of an individual person may be looked at again by someone of similar competence in that activity -a peer More formally it is a process of self-regulation by a profession or a process of evaluation involving qualified individuals within the relevant fields. Peer review methods are employed to maintain standards, improve performance and provide credibility.

### Technical Review

A technical review is a discussion meeting that focuses on achieving consensus about the technical content of a document. This is informal type of review so less process and documents are needed for this type of review. There is little or no focus on defect identification on the basis of referenced document, intended leadership and rules. During technical reviews defects are found by experts, who focus on the content of the document. The experts that are needed for a technical review are, for example, architects, chief designers and key users. No management participation required in technical review. More formal characteristics such as the use of checklists and logging list or issue log are optional. Informal review are commonly used and followed on regular basis in various organizations.

## 13.6 Formal Technical Reviews

A type of peer review, **”formal review”** follows a formal process and has a specific formal agenda. It has a well structured and regulated process, which is usually implemented at the end of each life cycle. During this process, a formal review panel or board considers the necessary steps for the next life cycle.

Features of Formal Review:

- This evaluates conformance to specification and various standards.
- Conducted by a group of 3 or more individuals.
- The review team petitions the management of technical leadership to act on the suggested recommendations.
- Here, the leader verifies that the action documents are verified and incorporated into external processes.
- Formal review consists of six important steps, which are:
    - Planning.
    - Kick-off.
    - Preparation.
    - Review meeting.
    - Rework.
    - Follow up.

### 13.6.1 Planning

The review process for a particular review begins with a ‘request for review’ by the author to the moderator (or inspection leader). A moderator is often assigned to take care of the scheduling (dates, time, place and invitation) of the review. On a project level, the project planning needs to allow time for review and rework activities, thus providing engineers with time to thoroughly participate in reviews.

For more formal reviews,e.g. inspections, the moderator always performs an entry check and defines at this stage formal exit criteria. The entry check is carried out to ensure that the reviewers' time is not wasted on a document that is not ready for review. A document containing too many obvious mistakes is clearly not ready to enter a formal review process and it could even be very harmful to the review process. It would possibly demotivate both reviewers and the author. Also, the review is most likely not effective because the numerous obvious and minor defects will conceal the major defects. Although more and other entry criteria can be applied, the following can be regarded as the minimum set for performing the entry check:

- A short check of a product sample by the moderator (or expert) does not reveal a large number of major defects.
- The document to be reviewed is available with line numbers.The document has been cleaned up by running any automated checks that apply.
- References needed for the inspection are stable and available.
- The document author is prepared to join the review team and feels confident with the quality of the document.

If the document passes the entry check, the moderator and author decide which part of the document to review. The maximum number of pages depends, among other things, on the objective, review type and document type and should be derived from practical experiences within the organization. For a review, the maximum size is usually between 10 and 20 pages. In formal inspection, only a page or two may be looked at in depth in order to find the most serious defects that are not obvious. After the document size has been set and the pages to be checked have been selected, the moderator determines, in cooperation with the author, the composition of the review team. The team normally consists of four to six participants, including moderator and author. To improve the effectiveness of the review, different roles are assigned to each of the participants. These roles help the reviewers focus on particular types of defects during checking. This reduces the chance of different reviewers finding the same defects. The moderator assigns the roles to the reviewers. Within reviews the following focuses can be identified:

- focus on higher-level documents, e.g. does the design comply to the requirements;
- focus on standards, e.g. internal consistency, clarity, naming conventions, templates;

- focus on related documents at the same level,
- focus on usage, e.g. for testability or maintainability.

### 13.6.2 Kick-off

An optional step in a review procedure is a kick-off meeting. The goal of this meeting is to get everybody on the same wavelength regarding the document under review and to commit to the time that will be spent on checking. Also the result of the entry check and defined exit criteria are discussed in case of a more formal review. In general a kick-off is highly recommended since there is a strong positive effect of a kick-off meeting on the motivation of reviewers and thus the effectiveness of the review process.

During the kick-off meeting the reviewers receive:

- A short introduction on the objectives of the review and the documents.
- The relationships between the document under review and the other documents (sources) are explained, especially if the number of related documents is high.

Role assignments, checking rate, the pages to be checked, process changes and possible other questions are also discussed during this meeting. Of course the distribution of the document under review, source documents and other related documentation, can also be done during the kick-off.

### 13.6.3 Preparation

The participants work individually on the document under review using:

- the related documents,
- procedures,
- rules and
- checklists provided.

The individual participants identify defects, questions and comments according to their understanding of the document and role. All issues are recorded, preferably using a logging form. The annotated document will be given to the author at the end of the logging meeting.

Using checklists during this phase can make reviews more effective and efficient. A critical success factor for a thorough preparation is the number of pages checked per hour. This is called the checking rate.

The optimum checking rate is the result of a mix of factors,

- including the type of document,
- its complexity,
- the number of related documents and
- the experience of the reviewer.

Usually the checking rate is in the range of five to ten pages per hour, but may be much less for formal inspection, e.g. one page per hour. During preparation, participants should not exceed this criterion. By collecting data and measuring the review process, company-specific criteria for checking rate and document size can be set, preferably specific to a document type.

## 13.6.4 Review meeting

The meeting typically consists of the following elements (partly depending on the review type):

- **Logging phase**

During the logging phase the issues, e.g. defects, that have been identified during the preparation are mentioned page by page, reviewer by reviewer and are logged either by the author or by a scribe. A separate person to do the logging (a scribe) is especially useful for formal review types such as an inspection. To ensure progress and efficiency, no real discussion is allowed during the logging phase. If an issue needs discussion, the item is logged and then handled in the discussion phase.

A detailed discussion on whether or not an issue is a defect is not very meaningful, as it is much more efficient to simply log it and proceed to the next one. Furthermore, in spite of the opinion of the team, a discussed and discarded defect may well turn out to be a real one during rework. Every defect and its severity should be logged. The participant who identifies the defect proposes the severity.

Severity classes could be:

- Critical: defects will cause downstream damage; the scope and impact of the defect is beyond the document under inspection.
- Major: defects could cause a downstream effect (e.g. a fault in a design can result in an error in the implementation).
- Minor: defects are not likely to cause downstream damage (e.g. non-compliance with the standards and templates).

In order to keep the added value of reviews, spelling errors are not part of the defect classification. Spelling defects are noted, by the participants, in the document under review and given to the author at the end of the meeting or could be dealt with in a separate proofreading exercise. During the logging phase the focus is on logging as many defects as possible within a certain time frame. To ensure this, the moderator tries to keep a good logging rate (number of defects logged per minute).

In a well-led and disciplined formal review meeting, the logging rate should be between one and two defects logged per minute. For a more formal review, the issues classified as discussion items will be handled during this meeting phase. Informal reviews will often not have a separate logging phase and will start immediately with discussion.

- **Discussion phase :** Participants can take part in the discussion by bringing forward their comments and reasoning. As chairman of the discussion meeting, the moderator takes care of people issues. Reviewers who do not need to be in the discussion may leave, or stay as a learning exercise. The moderator also paces this part of the meeting and ensures that all discussed items:

- either have an outcome by the end of the meeting, or
- are defined as an action point if a discussion cannot be solved during the meeting.

The outcome of discussions is documented for future reference.

- **Decision phase :** At the end of the meeting, a decision on the document under review has to be made by the participants, sometimes based on formal exit criteria. The most important exit criterion is the average number of critical and/or major defects found per page (e.g. no more than three critical/major defects per page).

If the number of defects found per page exceeds a certain level, the document must be reviewed again, after it has been reworked. If the document complies with the exit criteria, the document will be checked during follow-up by the moderator or one or more participants. Subsequently, the document can leave the review process.

If a project is under pressure, the moderator will sometimes be forced to skip re-reviews and exit with a defect-prone document. In addition to the number of defects per page, other exit criteria are used that measure the thoroughness of the review process, such as ensuring that all pages have been checked at the right rate. The average number of defects per page is only a valid quality indicator if these process criteria are met.

## 13.6.5 Rework

Based on the defects detected, the author will improve the document under review step by step. Not every defect that is found leads to rework. It is the author's responsibility to judge if a defect has to be fixed. If nothing is done about an issue for a certain reason, it should be reported to at least indicate that the author has considered the issue. Changes that are made to the document should be easy to identify during follow-up. Therefore the author has to indicate where changes are made.

## 13.6.6 Follow-up

The moderator is responsible for ensuring that satisfactory actions have been taken on all (logged) defects,process improvement suggestions and change requests. Although the moderator checks to make sure that the author has taken action on all known defects, it is not necessary for the moderator to check all the corrections in detail. If it is decided that all participants will check the updated document, the moderator takes care of the distribution and collects the feedback.

For more formal review types the moderator checks for compliance to the exit criteria. In order to control and optimize the review process, a number of measurements are collected by the moderator at each step of the process.

Examples of such measurements include:

- number of defects found
- number of defects found per page

- time spent checking per page
- total review effort, etc.

It is the responsibility of the moderator to ensure that the information is correct and stored for future analysis. Formal and informal review are two very important types of reviews that are used most commonly by software engineers to identify defects as well as to discuss ways to tackle these issues or discrepancies.

**Table - Formal Review Vs Informal Review**

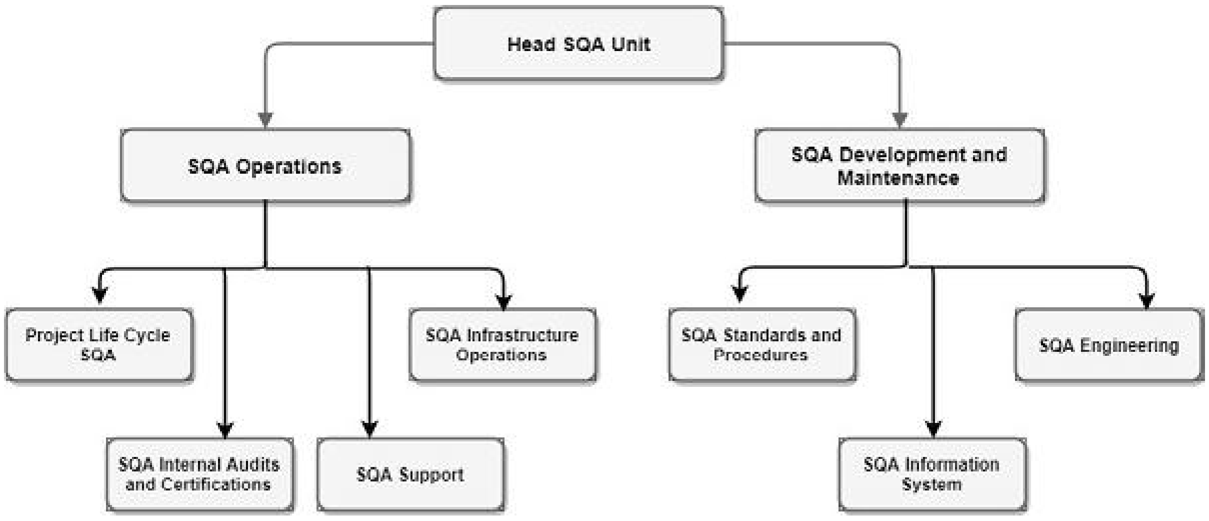
| Sl.No | Informal Review | Formal Review |
|---|---|---|
| 1 | Conducted on an as-needed I.e. Informal agenda | Conducted at the end of each life cycle phase I.e. Formal Agenda |
| 2 | The date and the time of the agenda for the Informal Review will not be addressed in the Project Plan | The agenda for the formal review must be addressed in the Project Plan |
| 3 | The developer chooses a review panel and provides and/or presents the material to be reviewed | The acquirer of the software appoints the formal review panel or board, who may make or affect a go/no-go decision to proceed to the next step of the life cycle. |
| 4 | The material may be as informal as a computer listing or hand-written documentation. | The material must be well prepared.<br><br>Ex Formal reviews include the Software Requirements Review, the Software Preliminary Design Review, the Software Critical Design Review, and the Software Test Readiness Review. |

## 13.7 Software Quality Assurance

Software Quality Assurance (SQA) is a set of activities for ensuring quality in software engineering processes. SQA is an ongoing process within the Software Development Life Cycle (SDLC) that routinely checks the developed software to ensure it meets the desired quality measures.

**SQA Tasks**

The structure of SQA unit varies by type and size of the organization. The following figure shows an example of a standard structure and all the components under an SQA unit. In this chapter, we will discuss the roles and responsibilities of each sub-unit.


Head SQA Unit
SQA Operations
SQA Development and Maintenance
Project Life Cycle SQA
SQA Infrastructure Operations
SQA Internal Audits and Certifications
SQA Support
SQA Standards and Procedures
SQA Engineering
SQA Information System

**SQA TASK**

❖ **Tasks Performed by the Head of the SQA Unit**

The head of the SQA unit is responsible for all the quality assurance tasks performed by the SQA unit and its sub-units. These tasks can be classified into the following categories "

- ➢ Planning tasks
- ➢ Management of the unit
- ➢ SQA professional activities

➢ **Planning Tasks**

- Preparation of the proposed annual activity program and budget for the unit
- Planning and updating the organization's software quality management system
- Preparation of the recommended annual SQA activities programs and SQA systems development plans for the software development and maintenance departments

- **Management Tasks**
  - Management of the SQA team's activities
  - Monitoring implementation of the SQA activity program
  - Nomination of team members, SQA committee members and SQA trustees
  - Preparation of special and periodic reports, e.g., the status of software quality issues within the organization and monthly performance reports
- **SQA Professional Activities**
  - Participation in project joint committees
  - Participation in formal design reviews
  - Review and approval of deviations from specifications
  - Consultation with project managers and team leaders
  - Participation in SQA committees and forums

❖ **Project Life Cycle SQA**

SQA tasks related to the project life cycle sub-unit may be classified into two groups "

- "Pure" managerial follow-up and approval tasks (project life cycle control tasks)
- "Hands-on" or active participation in project team SQA activities, where professional contributions are required (participation tasks)

❖ **Project Life Cycle Control Tasks**

- Follow-up of development and maintenance team's compliance with SQA procedures and work instructions
- Approval or recommendation of software products according to the relevant procedures
- Monitoring delivery of software maintenance services to internal and external customers

- Monitoring customer satisfaction and maintaining contact with customer's quality assurance representatives

## ❖ Participation Tasks

These tasks include participation in "

- Contract reviews
- Preparation and updating of project development and quality plans
- Formal design reviews
- Subcontractors' formal design reviews
- Software testing, including customer acceptance tests
- Software acceptance tests of subcontractors' software products
- Installation of new software products

## ❖ SQA Infrastructure Operations Tasks

SQA systems employ a variety of infrastructure components to operate smoothly, namely "

- Procedures and work instructions
- Supporting quality devices (templates, checklists)
- Staff training, instruction and certification
- Preventive and corrective actions
- Configuration management
- Documentation control

More specifically, the SQA sub-unit's tasks regarding these components include "

- Publication of updated versions of procedures, work instructions, templates, checklists, and so forth, together with their circulation in hard copy and/or by electronic means

- Transmission of training and instruction regarding adherence to and application of SQA procedures, work instructions and similar items to new and current staff
- Instruction of SQA trustees regarding new and revised procedures as well as development tools and methods, among other components
- Monitoring and supporting implementation of new and revised SQA procedures
- Follow-up of staff certification activities
- Proposal of subjects requiring preventive and corrective actions, including participation in CAB committees
- Follow-up of configuration management activities, including participation in CCA committees
- Follow-up of compliance with documentation procedures and work instruction**s**

❖ **SQA Internal Audit and Certification Tasks**

The types of SQA audits carried out in or by software organizations can be classified as follows“

- Internal audits
- Audits of subcontractors and suppliers to evaluate their SQA systems
- External audits performed by certification bodies
- External audits performed by customers who wish to evaluate the SQA system prior to accepting the organization as a supplier

The first two classes of audits are initiated and performed by the SQA subunit, the last two by external bodies.

SQA unit performs the following tasks for internal SQA audits

- Preparation of annual programs for internal SQA audits
- Performance of internal SQA audits

- Follow-up of corrections and improvements to be carried out by the audited teams and other units
- Preparation of periodic summary reports of the status of audit findings, including recommendations for improvements

SQA unit performs the following tasks for audits of subcontractors and suppliers “

- Preparation of the annual program for SQA audits of subcontractors and suppliers
- Performance of SQA audits of subcontractors and suppliers
- Follow-up of corrections and improvements to be carried out by the audited subcontractors and suppliers
- Collection of data on the performance of subcontractors and suppliers from internal as well as external sources
- Periodic evaluation of the organization’s certified subcontractors’ and suppliers’ SQA systems based on audit reports and information collected from other internal and external sources.

The evaluation report includes Recommendations regarding certification of subcontractors and suppliers

External audits performed by certification bodies involve the following tasks “

- Coordination of the certification audit’s contents and schedule
- Preparation of documents specified by the certification bodies
- Instruction of the audited teams and performance of the preparations necessary for certification audits
- Participation in certification audits
- Ensure required corrections and improvements are performed

SQA audits performed by the organization’s customers entail these tasks “

- Coordination of the audit’s contents and schedule
- Preparation of documents specified by the customer’s auditor
- Instruction of the audited teams and performance of the preparations necessary for SQA audits by the organization’s customers
- Participation in the audits
- Ensure that the required corrections and improvements are performed

❖ **SQA Support Tasks**

Most of the consumers of SQA support services are located within the organization. They include project managers, team leaders and SQA trustees. Their tasks include “

- Preparation of project plans and project quality plans
- Staffing review teams
- Choice of measures to solve identified software development risks
- Choice of measures to solve schedule delays and budget overruns
- Choice of SQA metrics and software costs components
- Use of SQA information system
- Choice of development methodologies and tools that reflect the failure experience data accumulated by the SQA unit

❖ **SQA Standards and Procedures Tasks**

The SQA sub-unit is intimately involved in deciding which SQA standards will be adopted as well as developing and maintaining the organization’s procedures. To fulfil the attendant obligations, the SQA unit is required to “

- Prepare an annual program for the development of new procedures and procedure updates

- Be responsible for the development of new procedures and procedure updates, with participation in appropriate committees and forums
- Follow-up on the developments and changes in SQA and software engineering standards; introduction of additional procedures and changes relevant to the organization
- Initiate updates and adaptations of procedures in response to changes in professional standards, including adoption or deletion of standards applied by the organization

❖ **SQA Engineering Tasks**

Follow-up of professional advances, solution of operational difficulties and expert analysis of failures are the immediate objectives of this SQA sub-unit.

Hence, the main engineering tasks involve the following “

- Testing quality and productivity aspects with respect to new development tools and new versions of currently used development tools
- Evaluation of quality and productivity of new development and maintenance methods and method improvements
- Development of solutions to difficulties confronted in application of currently used software development tools and methods
- Development of methods for measuring software quality and team productivity
- Provision of technological support to CAB committees during analysis of software development failures and formulation of proposed solutions

❖ **SQA Information Systems Tasks**

SQA information systems are meant to facilitate and improve the functioning of SQA systems. The tasks involved include “

- Development of SQA information systems for software development and maintenance units for

- o collection of activity data
- o processing of, for example, periodic reports, lists, exception reports and queries
- o processing of, for example, periodic reports, lists, exception reports and queries

- Development of SQA information systems facilitating the SQA unit's processing of information delivered by software development and maintenance units including estimates of software quality metrics and software quality costs
- Updating SQA information systems
- Development and maintenance of the organization's SQA Internet /Intranet site

❖ **SQA Trustees and Their Tasks**

SQA trustees are those members who are primarily involved in the promotion of software quality. These members provide the internal support necessary for successfully implementing SQA components.Their tasks may vary depending upon the organizations. Accordingly, it might be unit related and/or organization-related tasks.

❖ **Unit-related Tasks**

- Support colleagues for solving the difficulties during the implementation of software quality procedures and work instructions
- Assist the unit manager in carrying out related SQA tasks
- Promote compliance and monitor the implementation of SQA procedures and work instructions by colleagues
- Report substantial and systematic non-compliance events to the SQA unit
- Report severe software quality failures to the SQA unit

❖ **Organization-related Tasks**

- Trigger changes and updates of organization-wide SQA procedures and work instructions
- Trigger improvements of development and maintenance processes in the organization
- Initiate applications to the CAB regarding solutions to recurrent failures observed in the respective units
- Identify SQA training needs across the organization and propose appropriate training or instruction program to be conducted by the SQA unit

**SQA Committees and Their Tasks**

SQA committees can be either permanent or ad hoc. The tasks may vary considerably from organization to organization.

- Permanent committees commonly deal with SCC (Software Change Control), CA (Corrective Actions), procedures, method development tools and quality metrics.
- Ad hoc committees commonly deal with specific cases of general interest such as updating a specific procedure, analysis and solution of a software failure, elaborating software metrics for a targeted process or product, updating software quality costs and data collection methods for a specific issue.

Permanent SQA committees are integral parts of the SQA organizational framework; their tasks and operation are usually defined in the organization's SQA procedures. Ad hoc committees are established on a short-term per-problem basis, with members nominated by the executive responsible for software quality issues, the head of the SQA Unit, the SQA sub-units, permanent SQA committees, or any other body that initiated its formation and has an interest in the work. This body also defines the tasks of the ad hoc committee.

**Software Quality Assurance** (SQA) is a set of activities for ensuring quality in software engineering processes. It ensures that developed software meets and complies with the defined or standardized quality specifications. SQA is an ongoing process within the Software Development Life Cycle (SDLC) that routinely checks the developed software to ensure it meets the desired quality measures.

SQA practices are implemented in most types of software development, regardless of the underlying software development model being used. SQA incorporates and implements software testing methodologies to test the software. Rather than checking for quality after completion, SQA processes test for quality in each phase of development, until the software is complete. With SQA, the software development process moves into the next phase only once the current/ previous phase complies with the required quality standards. SQA generally works on one or more industry standards that help in building software quality guidelines and implementation strategies.

It includes the following activities “

- Process definition and implementation
- Auditing
- Training

Processes could be “

- Software Development Methodology
- Project Management
- Configuration Management
- Requirements Development/Management
- Estimation
- Software Design
- Testing, etc.

Once the processes have been defined and implemented, Quality Assurance has the following responsibilities “

- Identify the weaknesses in the processes
- Correct those weaknesses to continually improve the process

### 13.7.1 Components of SQA System

An SQA system always combines a wide range of SQA components. These components can be classified into the following six classes “

- **Pre-project components:** This assures that the project commitments have been clearly defined considering the resources required, the schedule and budget; and the development and quality plans have been correctly determined.

- **Components of project life cycle activities assessment:** The project life cycle is composed of two stages: the development life cycle stage and the operation–maintenance stage. The development life cycle stage components detect design and programming errors. Its components are divided into the following sub-classes: Reviews, Expert opinions, and Software testing.

The SQA components used during the operation–maintenance phase include specialized maintenance components as well as development life cycle components, which are applied mainly for functionality to improve the maintenance tasks.

- **Components of infrastructure error prevention and improvement:** The main objective of these components, which is applied throughout the entire organization, is to eliminate or at least reduce the rate of errors, based on the organization’s accumulated SQA experience.

- **Components of software quality management :** This class of components deals with several goals, such as the control of development and maintenance activities, and the introduction of early managerial support actions that mainly prevent or minimize schedule and budget failures and their outcomes.

- **Components of standardization, certification, and SQA system assessment :** These components implement international professional and managerial standards within the organization. The main objectives of this class are utilization of international professional knowledge, improvement of coordination of the organizational quality systems with other organizations, and assessment of the achievements of quality systems according to a common scale. The various standards may be classified into two main groups: quality management standards and project process standards.

**Organizing for SQA – the human components**

The SQA organizational base includes managers, testing personnel, the SQA unit and the persons interested in software quality such as SQA trustees, SQA committee members, and SQA forum members. Their main objectives are to initiate and support the implementation of SQA components, detect deviations from SQA procedures and methodology, and suggest improvements.

**Pre-project Software Quality Components**

These components help to improve the preliminary steps taken before starting a project. It includes "

- ❖ Contract Review
- ❖ Development and Quality Plans

❖ **Contract Review**

Normally, a software is developed for a contract negotiated with a customer or for an internal order to develop a firmware to be embedded within a hardware product. In all these cases, the development unit is committed to an agreed-upon functional specification, budget and schedule. Hence, contract review activities must include a detailed examination of the project proposal draft and the contract drafts.

Specifically, contract review activities include "

- Clarification of the customer's requirements
- Review of the project's schedule and resource requirement estimates
- Evaluation of the professional staff's capacity to carry out the proposed project
- Evaluation of the customer's capacity to fulfil his obligations
- Evaluation of development risks

❖ **Development and Quality Plans:** After signing the software development contract with an organization or an internal department of the same organization, a development plan of the project and its integrated quality assurance activities are prepared. These plans include additional details and needed revisions based on prior plans that provided the basis for the current proposal and contract. Most of the time, it takes several months between the tender submission and the signing of the contract. During these period, resources such as staff availability, professional capabilities may get changed. The plans are then revised to reflect the changes that occurred in the interim.

The main issues treated in the project development plan are “

- Schedules
- Required manpower and hardware resources
- Risk evaluations
- Organizational issues: team members, subcontractors and partnerships
- Project methodology, development tools, etc.
- Software reuse plans

The main issues treated in the project’s quality plan are “

- Quality goals, expressed in the appropriate measurable terms
- Criteria for starting and ending each project stage
- Lists of reviews, tests, and other scheduled verification and validation activities

## Check Your Progress

1. ____________is the process of evaluating the technical content and quality of the product.
2. The group size requires for conducting a informal review is ____ members.

   (a) 1-5 (b) 2-7 (c) 3-6 (d) 4-8
3. List out the purpose of Walkthrough.
4. Which of the following is optional step in formal review?

   (a) Planning (b) Preparation (c) Kick off (d) Rework

5. _________, ____________ and ________ are the severity classes in formal review.

6. Say True or False        Formal Review is conducted as and when needed in a project.

## 13.8 Goals and Metrics

Often when we talk about quality of software, we define it in terms of number of test cases, bugs, requirement coverage but this numbers in itself are not enough to incremental measure the quality of the application as well as quality of testing. The quality goals are owned by QA but defined by entire team and should be as important artifact as a test plan.

**What are Quality Goals**?

It is collection of individual metrics that helps to quantify quality of application as well as quality of testing.

**Why define and measure Quality Goals?**

Agile process gives QA as well as management to change or prioritize tasks at end of each iteration.The quality goal metrics accomplishes two things

- Establish protocol between product management and engineering what are the expectations from the product.
- Empower QA to have product management/development prioritize between new feature development vs fixing old bugs

**How to define Quality Goals:** The answer to above question will largely rely on processes in your individual institution, but I will take the lowest common denominator which I believe should be present in any QA infrastructure. The first part is to define ways to measure if the testing is sufficient.  If we are not doing the testing right, we lose the right to say that quality of application is sub-par.

**Quality Goals for Testing**

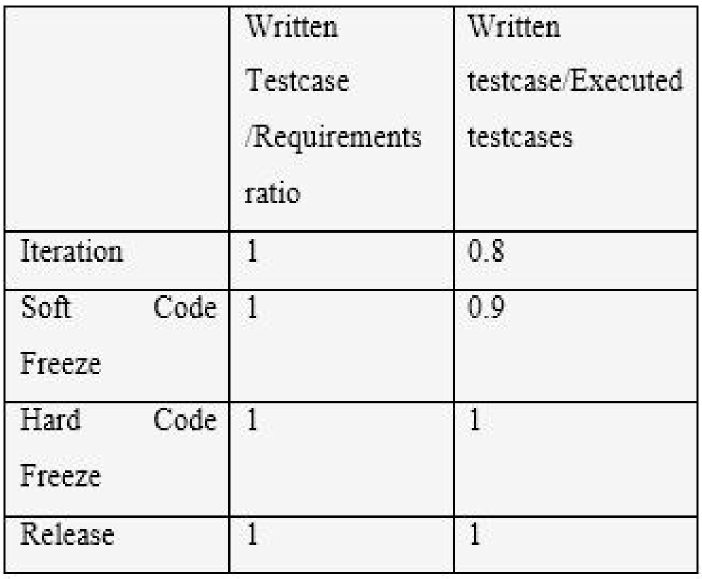
- Written Test case /Requirements ratio: This number should be greater than 1. Of course this assumes that requirements are complete and sufficient. So for this discussion let’s assume that all the requirements that are known to entire team should qualify for this metrics.
- Bugs yet to be verified

**Quality Goals for Application**

- Non-executed test cases: The number to achieve is 0.
- High severity Open Bugs/Total Bugs
- Medium severity Open Bugs/Total Bugs
- Low severity Open Bugs/Total Bugs
- Untargeted Bugs

The above metrics will vary depending on your own process and outputs that QA efforts produce. You can define it in terms of iteration, release milestones or combination of both. E.g it can look like following. The following figure shows quality goals for measuring quality of testing.

**Quality Goals**



| | Written Testcase /Requirements ratio | Written testcase/Executed testcases |
|---|---|---|
| Iteration | 1 | 0.8 |
| Soft Code Freeze | 1 | 0.9 |
| Hard Code Freeze | 1 | 1 |
| Release | 1 | 1 |

The idea is to measure this against actual numbers and depending on that take corrective action to either achieve the goal or lower the goal. The more often we measure this, we have the option to do something about quality of application.

## 13.9 Summary

From the above discussion, we can conclude that software review is a vital part of "software development life cycle (SDLC)" that helps developers and other members related to the project, to improve the quality and other components of the software. With the assistance of software review, a team can effortlessly identify and resolve issues from the software product, while meeting its requirements, standards, and other necessary criteria. In short, software review is a helpful process that allows developers to deliver a quality product to the client.

## 13.10 Check Your Answers

1. Peer review
2. b. 2-7
3. First Problems, Discuss alternative solutions and focusing on demonstrating how work product meets all requirements.
4. c. Kickoff
5. Critical, Major and Minor
6. False

## 13.11 Model Questions

1. What is the need for software review?
2. Explain in detail about the types of software review.
3. What is informal reviews and explain its types.
4. Explain the steps in formal review.
5. What is Software Quality Assurance?
6. Explain the Software Quality Assurance Tasks.
7. What are Quality Goals?
8. What are the Quality goals for testing and application?

# LESSON-14

# SOFTWARE RELIABILITY

## 14.1 Learning Objective

Upon successful completion of this lesson, you will be able to:

- Define what Reliability is.
- Describe what unit Testing is.
- Understand the Integration Testing.

### Structure

## 14.2 Introduction

Software Reliability is not a function of time - although researchers have come up with models relating the two. The modeling technique for Software Reliability is reaching its prosperity, but before using the technique, we must carefully select the appropriate model that can best suit our case. Measurement in software is still in its infancy. No good quantitative methods have been developed to represent Software Reliability without excessive limitations. Various approaches can be used to improve the reliability of software, however, it is hard to balance development time and budget with software reliability.

## 14.3 Software Reliability

Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability. It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection. The high complexity of software is the major contributing factor of Software Reliability problems.

A good software reliability engineering program, introduced early in the development cycle, will mitigate these problems by: Preparing program management in advance for the testing effort and allowing them to plan both schedule and budget to cover the required testing.

If requirements are incomplete there will be no testing of the exception conditions.

We can support or lead tasks such as:

1) Reliability Allocation

2) Defining and Analyzing Operational Profiles

3) Test Preparation and Plan

4) Software Reliability Models

**Reliability Allocation**:-

Reliability allocation is the task of defining the necessary reliability of a software item. The item may be part of an integrated hardware/software system, may be a relatively independent software application, or, more and more rarely, a standalone software program. In either of these cases our goal is to bring system reliability within either a strict constraint required by a customer or an internally perceived readiness level, or optimize reliability within schedule and cost constraints.

**Defining and Analyzing Operational Profiles**:-

The reliability of software, much more so than the reliability of hardware, is strongly tied to the operational usage of an application. A software fault may lead to system failure only if that fault is encountered during operational usage. If a fault is not accessed in a specific operational mode, it will not cause failures at all. It will cause failure more often if it is located in code that is part of an often used "operation" (An operation is defined as a major logical task, usually repeated multiple times within an hour of application usage). Therefore in software reliability engineering we focus on the operational profile of the software which weighs the occurrence probabilities of each operation. Unless safety requirements indicate a modification of this approach we will prioritize our testing according to this profile.

**Test Preparation and Plan**:-

Test preparation is a crucial step in the implementation of an effective software reliability program. A test plan that is based on the operational profile on the one hand, and subject to the reliability allocation constraints on the other, will be effective at bringing the program to its reliability goals in the least amount of time and cost.

Software Reliability Engineering is concerned not only with feature and regression test, but also with load test and performance test. All these should be planned based on the activities outlined above.

The reliability program will inform and often determine the following test preparation activities:

- Assessing the number of new test cases required for the current release.
- New test case allocation among the systems (if multi-system).
- New test case allocation for each system among its new operations.
- Specifying new test cases
- Adding the new test cases to the test cases from previous releases.

**Software Reliability Models**:-

Software reliability engineering is often identified with reliability models, in particular reliability growth models. These, when applied correctly, are successful at providing guidance to management decisions such as:

- Test schedule
- Test resource allocation

- Time to market
- Maintenance resource allocation

## 14.4 A Strategic Approach to Software Testing

Software testing is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is defect free. It involves execution of a software component or system component to evaluate one or more properties of interest.

Software testing also helps to identify errors, gaps or missing requirements in contrary to the actual requirements. It can be either done manually or using automated tools. Some prefer saying Software testing as a White Box and Black Box Testing.

In simple terms, Software Testing means Verification of Application under Test (AUT).

Why is Software Testing Important?

Testing is important because software bugs could be expensive or even dangerous. Software bugs can potentially cause monetary and human loss, and history is full of such examples.

- In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.
- Nissan cars have to recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.
- Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point store served coffee for free as they unable to process the transaction.
- Some of the Amazon's third party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.
- Vulnerability in Window 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.

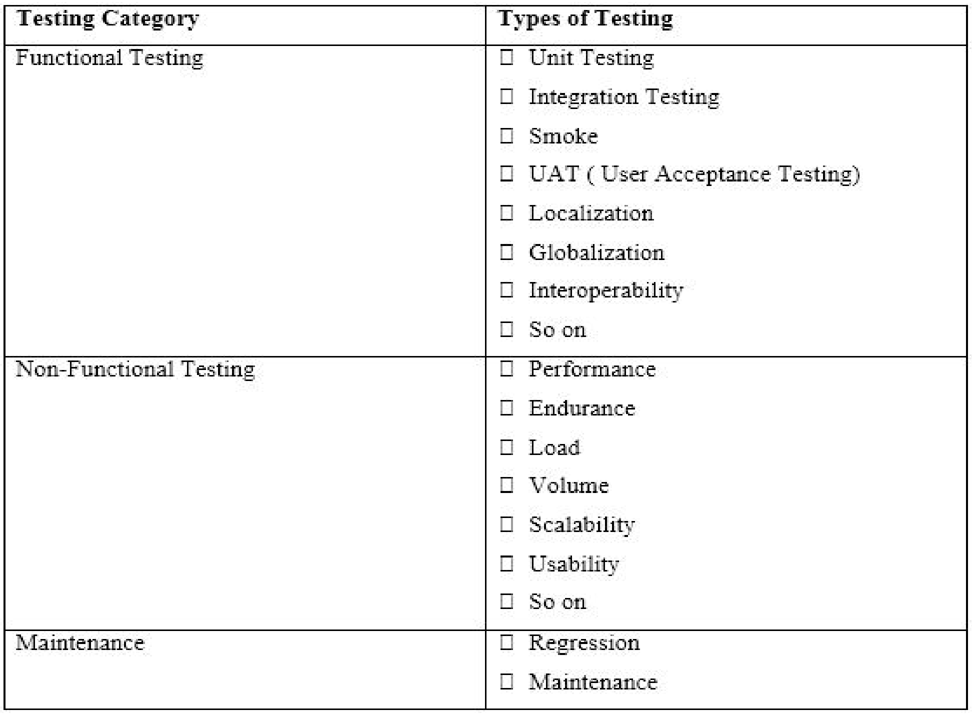
- In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocent live
- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.
- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
- In May 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.

**Testing Categories and its types**

Typically Testing is classified into three categories.

- Functional Testing
- Non-Functional Testing or Performance Testing
- Maintenance (Regression and Maintenance)

**Table - Testing Categories and its types**



| Testing Category | Types of Testing |
|---|---|
| Functional Testing | □ Unit Testing<br>□ Integration Testing<br>□ Smoke<br>□ UAT ( User Acceptance Testing)<br>□ Localization<br>□ Globalization<br>□ Interoperability<br>□ So on |
| Non-Functional Testing | □ Performance<br>□ Endurance<br>□ Load<br>□ Volume<br>□ Scalability<br>□ Usability<br>□ So on |
| Maintenance | □ Regression<br>□ Maintenance |

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by it’s design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

Software testing can be divided into two steps:

1. Verification: it refers to the set of tasks that ensure that software correctly implements a specific function.

2. Validation: it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: ”Are we building the product right?”

Validation: ”Are we building the right product?”

What are different types of software testing?

Software Testing can be broadly classified into two types:

1. Manual Testing: Manual testing includes testing a software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behavior or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing.

Testers use test plans, test cases, or test scenarios to test a software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.

2. Automation Testing: Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly.

Apart from regression testing, automation testing is also used to test the application from load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money in comparison to manual testing.

What are different techniques of Software Testing?

Software techniques can be majorly classified into two categories:

1. Black Box Testing: The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without concerning with the internal logical structure of the software is known as black box testing.

2. White-Box Testing: The technique of testing in which the tester is aware of the internal workings of the product, have access to it's source code and is conducted by making sure that all internal operations are performed according to the specifications is known as white box testing.

**Table - Difference between Black Box and White Box Testing**

| **BLACK BOX TESTING** | **WHITE BOX TESTING** |
|---|---|
| Internal workings of an application are not required. | Knowledge of the internal workings is must. |
| Also known as closed box/data driven testing. | Also known as clear box/structural testing. |
| End users, testers and developers. | Normally done by testers and developers. |
| This can only be done by trial and error method. | Data domains and internal boundaries can be better tested. |

What are different levels of software testing?

Software level testing can be majorly classified into 4 levels:

1. Unit Testing: A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.

2. Integration Testing: A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

3. System Testing: A level of the software testing process where a complete, integrated system/ software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.

4. Acceptance Testing: A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

## 14.5 Unit Testing

**UNIT TESTING** is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class. (Some treat a module of an application as a unit. This is to be discouraged as there will probably be many individual units within that module.) Unit testing frameworks, drivers, stubs, and mock/ fake objects are used to assist in unit testing.

**Unit Testing Method**

It is performed by using the White Box Testing method.

**When is it performed?**

Unit Testing is the first level of software testing and is performed prior to Integration Testing.

**Who performs it?**

It is normally performed by software developers themselves or their peers. In rare cases, it may also be performed by independent software testers.

**Unit Testing Tasks**

- Unit Test Plan
    - Prepare
    - Review

  - Rework
  - Baseline
- Unit Test Cases/Scripts
  - Prepare
  - Review
  - Rework
  - Baseline
- Unit Test
  - Perform

**Unit Testing Benefits**

- Unit testing increases confidence in changing/ maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change. Also, if codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.
- Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.
- Development is faster. How? If you do not have unit testing in place, you write your code and perform that fuzzy 'developer test' (You set some breakpoints, fire up the GUI, provide a few inputs that hopefully hit your code and hope that you are all set.) But, if you have unit testing in place, you write the test, write the code and run the test. Writing tests takes time but the time is compensated by the less amount of time it takes to run the tests; You need not fire up the GUI and provide all those inputs. And, of course, unit tests are more reliable than 'developer tests'. Development is faster in the long run too. How? The effort required to find and fix defects found during unit testing is very less in comparison to the effort required to fix defects found during system testing or acceptance testing.

- The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels. Compare the cost (time, effort, destruction, humiliation) of a defect detected during acceptance testing or when the software is live.
- Debugging is easy. When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days/weeks/ months need to be scanned.
- Codes are more reliable. Why? I think there is no need to explain this to a sane person.

**The reason behind Unit Testing**

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

**Some crucial reasons are listed below:**

- Unit testing helps tester and developers to understand the base of code that makes them able to change defect causing code quickly.
- Unit testing helps in the documentation.
- Unit testing fixes defects very early in the development phase that's why there is a possibility to occur a smaller number of defects in upcoming testing levels.
- It helps with code reusability by migrating code and test cases.

**Unit Testing Techniques:**

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

- Data flow Testing
- Control Flow Testing
- Branch Coverage Testing

- Statement Coverage Testing
- Decision Coverage Testing

**Unit Testing Tools:**

**NUnit:** NUnit is a unit testing framework used mainly for.Net languages. It allows scripting of test cases manually and supports data-driven test cases.

**JUnit:** JUnit is a unit testing framework used mainly for java languages. JUnit provides assertions for the identification of methods.

**PHP Unit:** PHPUnit unit testing tool used for PHP language. It provides assertions to use assertion methods (Methods are pre-defined) to make sure that system behaves in a required manner.

**ParasoftJtest:** ParasoftJtest is an integrated IDE plugin Junit, Mockito, PowerMock, and Spring with easy one-click activities for scaling, creating and maintaining unit testing.

**EMMA:** EMMA is an open source unit testing tool used for java language coding. It analyzes and reports the code of java language.

**How to achieve the best result via Unit testing?**

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

- Test cases must be independent because if there is any change or enhancement in requirement, the test cases will not be affected.
- Naming conventions for unit test cases must be clear and consistent.
- During unit testing, the identified bugs must be fixed before jump on next phase of the SDLC.
- Only one code should be tested at one time.
- Adopt test cases with the writing of the code, if not doing so, the number of execution paths will be increased.
- If there are changes in the code of any module, ensure the corresponding unit test is available or not for that module.

**Advantages of Unit Testing**

- Unit testing uses module approach due to that any part can be tested without waiting for completion of another parts testing.
- The developing team focuses on the provided functionality of the unit and how functionality should look in unit test suits to understand the unit API.
- Unit testing allows the developer to refactor code after a number of days and ensure the module still working without any defect.

**Disadvantages of Unit Testing**

- It cannot identify integration or broad level error as it works on units of the code.
- In the unit testing, evaluation of all execution paths is not possible, so unit testing is not able to catch each and every error in a program.
- It is best suitable for conjunction with other testing activities.

## 14.6 Integration Testing

**INTEGRATION TESTING** is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration Testing.

- **integration testing:** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
- **component integration testing:** Testing performed to expose defects in the interfaces and interaction between integrated components.
- **system integration testing:** Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet).

**Analogy**

During the process of manufacturing a ballpoint pen, the cap, the body, the tail and clip, the ink cartridge and the ballpoint are produced separately and unit tested separately. When two or more units are ready, they are assembled and Integration Testing is performed. For example, whether the cap fits into the body or not.

**Method**

Any of Black Box Testing, White Box Testing and Gray Box Testing methods can be used. Normally, the method depends on your definition of ‘unit’.

Tasks

- Integration Test Plan
    - Prepare
    - Review
    - Rework
    - Baseline
- Integration Test Cases/Scripts
    - Prepare
    - Review
    - Rework
    - Baseline
- Integration Test
    - Perform

**When is Integration Testing performed?**

Integration Testing is the second level of testing performed after Unit Testing and before System Testing.

**Who performs Integration Testing?**

Developers themselves or independent testers perform Integration Testing.

**Approaches**

- *Big Bang* is an approach to Integration Testing where all or most of the units are combined together and tested at one go. This approach is taken when the testing team receives the entire software in a bundle. So what is the difference between Big Bang Integration Testing and System Testing? Well, the former tests only the interactions between the units while the latter tests the entire system.
- *Top Down* is an approach to Integration Testing where top-level units are tested first and lower level units are tested step by step after that. This approach is taken when top-down development approach is followed. Test Stubs are needed to simulate lower level units which may not be available during the initial phases.
- *Bottom Up* is an approach to Integration Testing where bottom level units are tested first and upper-level units step by step after that. This approach is taken when bottom-up development approach is followed. Test Drivers are needed to simulate higher level units which may not be available during the initial phases.
- *Sandwich/Hybrid* is an approach to Integration Testing which is a combination of Top Down and Bottom Up approaches.

Tips

- Ensure that you have a proper Detail Design document where interactions between each unit are clearly defined. In fact, you will not be able to perform Integration Testing without this information.
- Ensure that you have a robust Software Configuration Management system in place. Or else, you will have a tough time tracking the right version of each unit, especially if the number of units to be integrated is huge.
- Make sure that each unit is unit tested before you start Integration Testing.
- As far as possible, automate your tests, especially when you use the Top Down or Bottom Up approach, since regression testing is important each time you integrate a unit, and manual regression testing can be inefficient.

Although each software module is unit tested, defects still exist for various reasons like

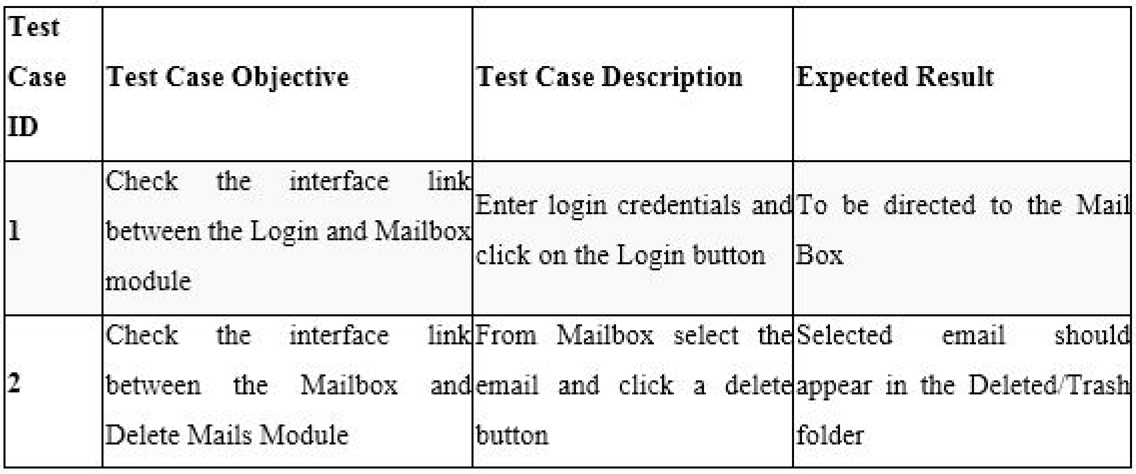
- A Module, in general, is designed by an individual software developer whose understanding and programming logic may differ from other programmers. Integration Testing becomes necessary to verify the software modules work in unity
- At the time of module development, there are wide chances of change in requirements by the clients. These new requirements may not be unit tested and hence system integration Testing becomes necessary.
- Interfaces of the software modules with the database could be erroneous
- External Hardware interfaces, if any, could be erroneous
- Inadequate exception handling could cause issues.

**Integration Test Case**

Integration Test Case differs from other test cases in the sense it **focuses mainly on the interfaces & flow of data/information between the modules**. Here priority is to be given for the **integrating links** rather than the unit functions which are already tested.

Sample Integration Test Cases for the following scenario: Application has 3 modules say ‘Login Page’, ‘Mailbox’ and ‘Delete emails’ and each of them is integrated logically. Here do not concentrate much on the Login Page testing as it’s already been done in Unit Testing. But check how it’s linked to the Mail Box Page. Similarly Mail Box: Check its integration to the Delete Mails Module.

**Table - Integration Test Case**



| Test Case ID | Test Case Objective | Test Case Description | Expected Result |
|---|---|---|---|
| 1 | Check the interface link between the Login and Mailbox module | Enter login credentials and click on the Login button | To be directed to the Mail Box |
| 2 | Check the interface link between the Mailbox and Delete Mails Module | From Mailbox select the email and click a delete button | Selected email should appear in the Deleted/Trash folder |

### Check Your Progress

1. __________is the task of defining the necessary reliability of a software item.

2. List out the two steps in software testing.

3. Which of the following is a open source unit testing tool for java language coding.

(a). EMMA (b) NUnit (c) JUnit (d)PeoplesoftJtest

4. Say True or False Internal workings of an application are not required in Black Box Testing.

5. List out the approaches in Integration Testing.

## 14.7 Approaches/Methodologies/Strategies of Integration Testing

Software Engineering defines variety of strategies to execute Integration testing, viz.

- Big Bang Approach :
- Incremental Approach: which is further divided into the following
    - Top Down Approach
    - Bottom Up Approach
    - Sandwich Approach - Combination of Top Down and Bottom Up

Below are the different strategies, the way they are executed and their limitations as well advantages.

**Big Bang Approach:** In this approach, testing is done via integration of all modules at once. It is convenient for small software systems, if used for large software systems identification of defects is difficult.Since this testing can be done after completion of all modules due to that testing team has less time for execution of this process so that internally linked interfaces and high-risk critical modules can be missed easily.Here all component are integrated together at **once** and then tested.

**Advantages:**

- Convenient for small systems.

**Disadvantages:**

- Fault Localization is difficult.
- Given the sheer number of interfaces that need to be tested in this approach, some interfaces link to be tested could be missed easily.
- Since the Integration testing can commence only after "all" the modules are designed, the testing team will have less time for execution in the testing phase.
- Since all modules are tested at once, high-risk critical modules are not isolated and tested on priority. Peripheral modules which deal with user interfaces are also not isolated and tested on priority.
- Identification of defect is difficult.
- Small modules missed easily.
- Time provided for testing is very less.

**Incremental Approach**

In this approach, testing is done by joining two or more modules that are ***logically related***. Then the other related modules are added and tested for the proper functioning. The process continues until all of the modules are joined and tested successfully.

Incremental Approach, in turn, is carried out by two different Methods:

- Bottom Up
- Top Down

**What is Stub and Driver?**

Incremental Approach is carried out by using dummy programs called **Stubs and Drivers**. Stubs and Drivers do not implement the entire programming logic of the software module but just simulate data communication with the calling module.

**Stub**: Is called by the Module under Test.

**Driver**: Calls the Module to be tested.

**Bottom up Integration Approach**

In the bottom-up strategy, each module at lower levels is tested with higher modules until all modules are tested. It takes help of Drivers for testing

**Diagrammatic Representation**:

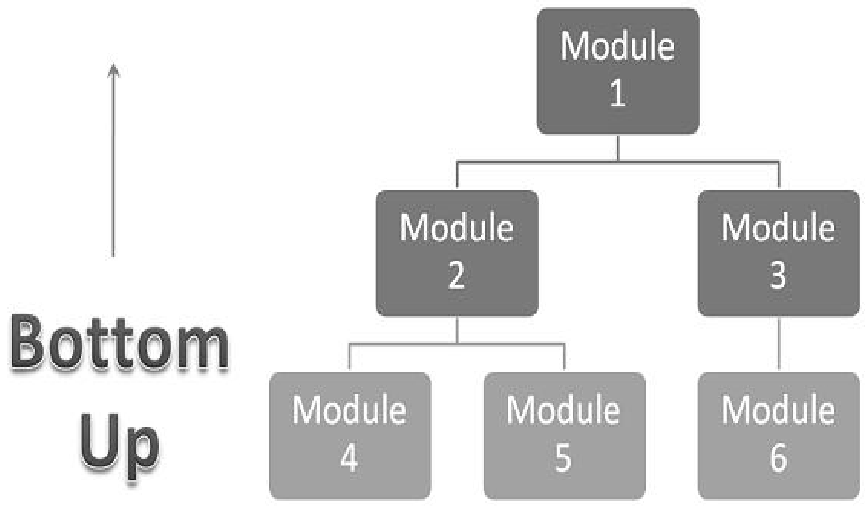


**Figure Bottom Up Integration Approach**

**Advantages:**

- Fault localization is easier.
- No time  is wasted waiting for all modules to be developed unlike Big-bang approach

**Disadvantages:**

- Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
- An early prototype is not possible

**Top-down Integration:**

In Top to down approach, testing takes place from top to down following the control flow of the software system.

Takes help of stubs for testing.

**Bottom up Integration Approach**

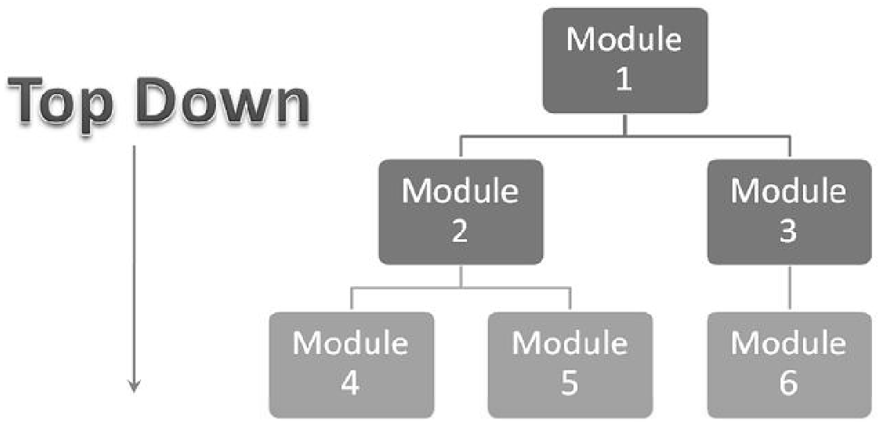


**Figure - Top Down Integration Approach**

**Advantages:**

- Fault Localization is easier.
- Possibility to obtain an early prototype.
- Critical Modules are tested on priority; major design flaws could be found and fixed first.

**Disadvantages:**

- Needs many Stubs.
- Modules at a lower level are tested inadequately.

**Hybrid/ Sandwich Integration**

In the sandwich/hybrid strategy is a combination of Top Down and Bottom up approaches. Here, top modules are tested with lower modules at the same time lower modules are integrated with top modules and tested. This strategy makes use of stubs as well as drivers.

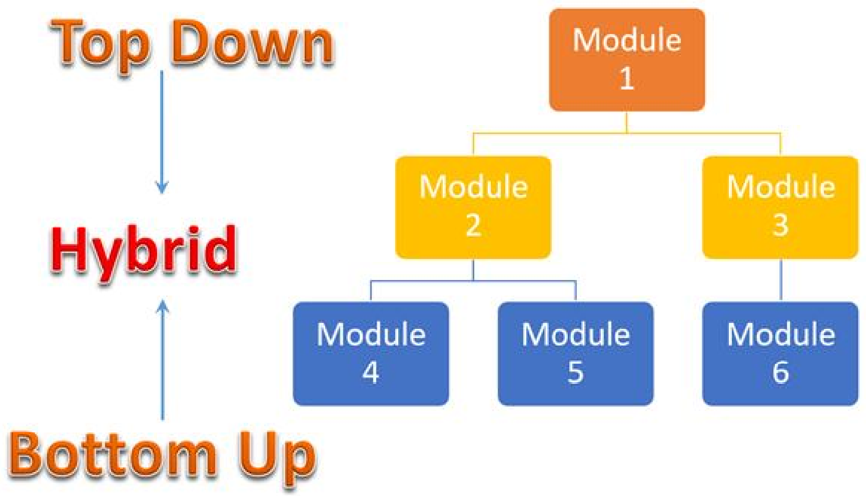


**Figure - Hybrid Integration Approach**

**How to do Integration Testing?**

The Integration test procedure irrespective of the Software testing strategies (discussed above):

1. Prepare the Integration Tests Plan
2. Design the Test Scenarios, Cases, and Scripts.
3. Executing the test Cases followed by reporting the defects.
4. Tracking & re-testing the defects.
5. Steps 3 and 4 are repeated until the completion of Integration is successful.

**Brief Description of Integration Test Plans:**

It includes the following attributes:

- Methods/Approaches to testing (as discussed above).
- Scopes and Out of Scopes Items of Integration Testing.

- Roles and Responsibilities.
- Pre-requisites for Integration testing.
- Testing environment.
- Risk and Mitigation Plans.

**Entry and Exit Criteria of Integration Testing**

Entry and Exit Criteria to Integration testing phase in any software development model

**Entry Criteria:**

- Unit Tested Components/Modules
- All High prioritized bugs fixed and closed
- All Modules to be code completed and integrated successfully.
- Integration tests Plan, test case, scenarios to be signed off and documented.
- Required Test Environment to be set up for Integration testing

**Exit Criteria:**

- Successful Testing of Integrated Application.
- Executed Test Cases are documented
- All High prioritized bugs fixed and closed
- Technical documents to be submitted followed by release Notes.

**Best Practices/ Guidelines for Integration Testing**

- First, determine the Integration Test Strategy that could be adopted and later prepare the test cases and test data accordingly.
- Study the Architecture design of the Application and identify the Critical Modules. These need to be tested on priority.
- Obtain the interface designs from the Architectural team and create test cases to verify all of the interfaces in detail. Interface to database/external hardware/software application must be tested in detail.

- After the test cases, it’s the test data which plays the critical role.
- Always have the mock data prepared, prior to executing. Do not select test data while executing the test cases.

## 14.8 Summary

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use.

## 14.9 Check Your Answers

1. Reliability allocation.
2. Verification and validation.
3. (a) EMMA
4. True
5. Big Bang and Incremental Approach.

## 14.10 Model Questions

1. Define Unit Testing.
2. What is Black Box Testing?
3. Differentiate between Black Box and White Box Testing.
4. Explain the Approaches for Unit Testing.
5. Describe about the tools available for unit testing.
6. Explain the Integration Testing and its techniques.
7. What is stub and driver?
8. What are the benefits of Unit Testing?
9. List out the advantages and disadvantages of Unit Testing.
10. List out the advantages and disadvantages of Integration Testing.

# LESSON -15

# VALIDATION TESTING

## 15.1 Learning Objective

Upon successful completion of this lesson, you will be able to:

- Understand what is the need for validation Testing.
- Describe the importance of Acceptance Testing.
- Know the need for software maintenance.
- Identify the necessity of software debugging.

## Structure

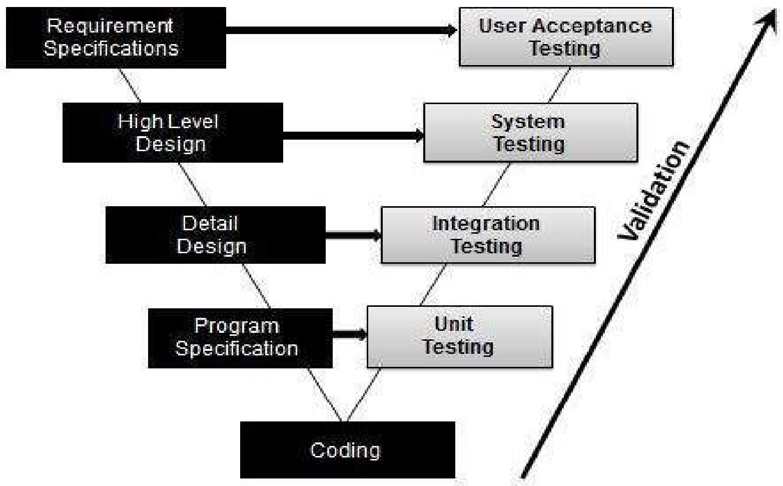
## 15.2 Introduction

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements. Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

### 15.3 Validation Testing

**Validation Testing - Workflow**

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



**Activities**

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

A product can pass while verification, as it is done on the paper and no running or functional application is required. But, when same points which were verified on the paper is actually developed then the running application or product can fail while validation. This may happen because when a product or application is build as per the specification but these specifications are not up to the mark hence they fail to address the user requirements.

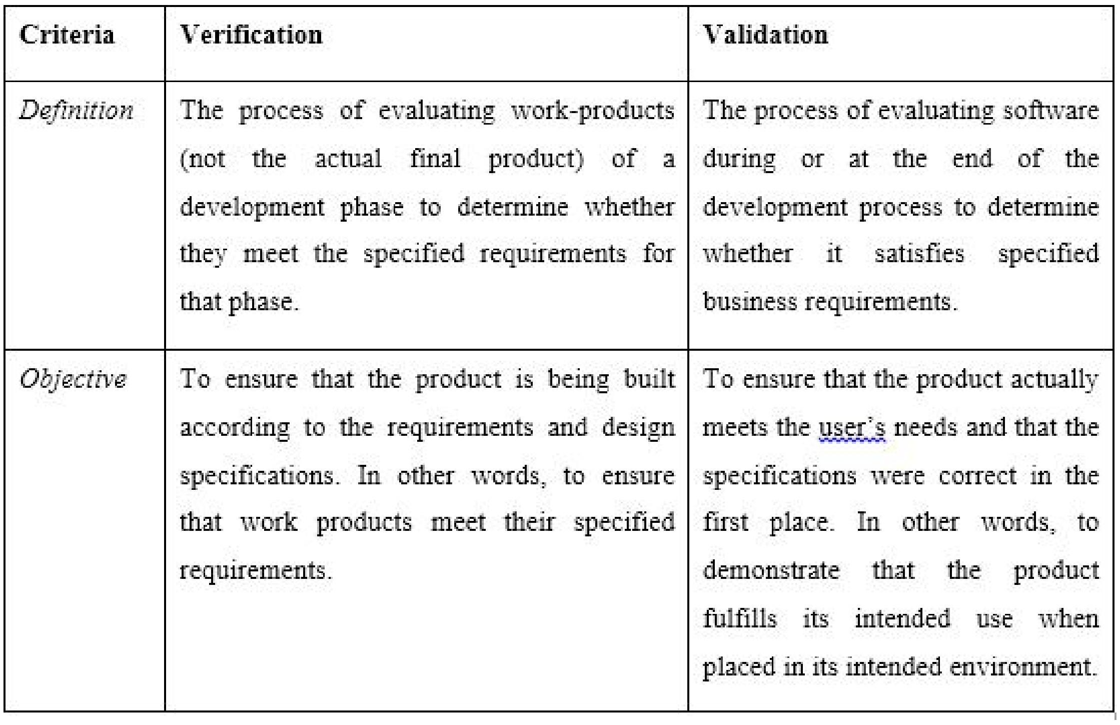
**Advantages of Validation**

1. During verification if some defects are missed then during validation process it can be caught as failures.
2. If during verification some specification is misunderstood and development had happened then during validation process while executing that functionality the difference between the actual result and expected result can be understood.
3. Validation is done during testing like feature testing, integration testing, system testing, load testing, compatibility testing, stress testing, etc.
4. Validation helps in building the right product as per the customer's requirement and helps in satisfying their needs.

Validation is basically done by the testers during the testing. While validating the product if some deviation is found in the actual result from the expected result then a bug is reported or an incident is raised. Not all incidents are bugs. But all bugs are incidents. Incidents can also be of type 'Question' where the functionality is not clear to the tester.

Hence, validation helps in unfolding the exact functionality of the features and helps the testers to understand the product in much better way. It helps in making the product more user friendly. **VERIFICATION vs VALIDATION** are hugely confused and debated terms in the software testing world. You will encounter (or have encountered) all kinds of usage and interpretations of these terms, and it is our humble attempt here to distinguish between them as clearly as possible.

**Table - Difference between Verification and Validation**



| **Criteria** | **Verification** | **Validation** |
|---|---|---|
| *Definition* | The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase. | The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements. |
| *Objective* | To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements. | To ensure that the product actually meets the user's needs and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment. |

## 15.4 Acceptance testing

Acceptance testing is formal testing based on user requirements and function processing. It determines whether the software is conforming specified requirements and user requirements or not. It is conducted as a kind of Black Box testing where the number of required users involved testing the acceptance level of the system. It is the fourth and last level of software testing.

However, the software has passed through three testing levels (Unit Testing, Integration Testing, System Testing). But still there are some minor errors which can be identified when the system is used by the end user in the actual scenario. Acceptance testing is the squeezing of all the testing processes that have done previously.

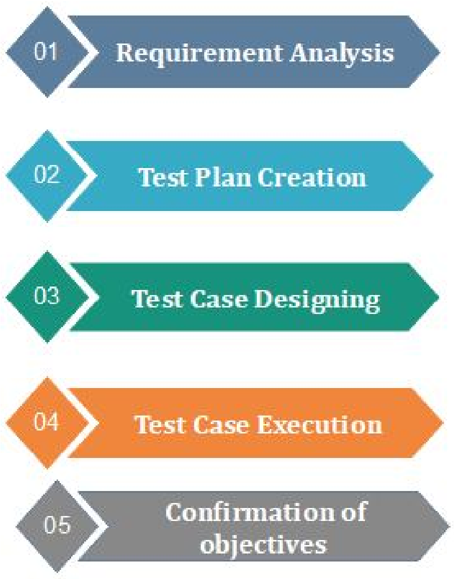
### Reason behind Acceptance Testing

Once the software has undergone through Unit Testing, Integration Testing and System Testing so, Acceptance Testing may seem redundant, but it is required due to the following reasons.

- During the development of a project if there are changes in requirements and it may not be communicated effectively to the development team.

- Developers develop functions by examining the requirement document on their own understanding and may not understand the actual requirements of the client.
- There may be some minor errors which can be identified only when the system is used by the end user in the actual scenario so, to find out these minor errors, acceptance testing is essential.

**Steps to Perform Acceptance Testing**



**Requirement Analysis**

In this step, the testing team analyzes requirement document to find out the objective of the developed software. Test planning accomplished by using requirement document, Process Flow Diagrams, System Requirements Specification, Business Use Cases, Business Requirements Document and Project Charter.

**Test Plan Creation**

Test Plan Creation outlines the whole strategy of the testing process. This strategy is used to ensure and verify whether the software is conforming specified requirements or not.

**Test Case Designing**

This step includes the creation of test cases based on test plan documents. Test cases should be designed in a way that can cover most of the acceptance testing scenario.

**Test Case Execution**

Test Case Execution includes execution of test cases by using appropriate input values. The testing team collects input values from the end user then all test cases are executed by both tester and end user to make sure software is working correctly in the actual scenario.

**Confirmation of objectives**

After successful completion of all testing processes, testing team confirms that the software application is bug-free and it can be delivered to the client.

**Tools used in Acceptance Testing**

Acceptance Testing can be done by using several tools; some are given below:

**Watir:**

Acceptance testing uses this tool for the execution of automated browser-based test cases. It uses Ruby language for the inter-process communication.

**Fitness tool:**

This tool is used to enter input values and generate test cases automatically. The user needs to input values, these values used by the tool to execute test cases and to produce output. It uses Java language for the inter-process communication. This tool makes it easy to create test cases as well as record them in the form of a table.

**Advantages of Acceptance Testing**

- It increases the satisfaction of clients as they test application itself.
- The quality criteria of the software are defined in an early phase so that the tester has already decided the testing points. It gives a clear view to testing strategy.
- The information gathered through acceptance testing used by stakeholders to better understand the requirements of the targeted audience.
- It improves requirement definition as client tests requirement definition according to his needs.

**Disadvantages of Acceptance Testing**

According to the testing plan, the customer has to write requirements in their own words and by themselves but

a. Customers are not willing to do that; it defeats the whole point of acceptance testing.

b. If test cases are written by someone else, the customer does not understand them, so tester has to perform the inspections by themselves only.

If the process is done in this manner, it destroys the existence of the Acceptance Testing.

## 15.5 System Testing

**SYSTEM TESTING** is a level of software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements. System Testing (ST) is a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements. In System testing, the functionalities of the system are tested from an end-to-end perspective. System Testing is usually carried out by a team that is independent of the development team in order to measure the quality of the system unbiased. It includes both functional and Non-Functional testing.

**Types of System Test**

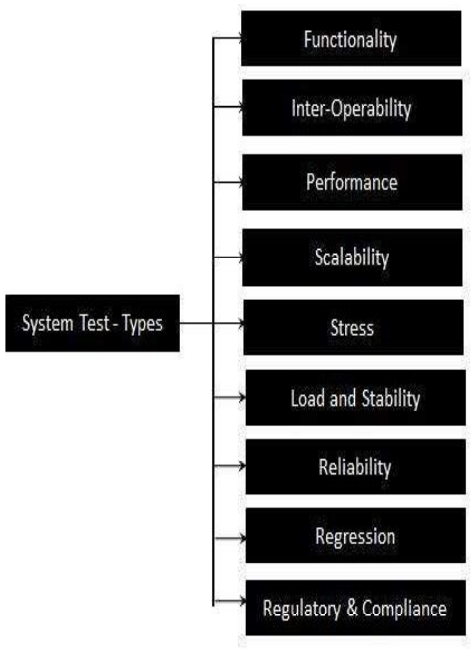


**Figure System Test Types**

System testing, also referred to as system-level tests or system-integration testing, is the process in which a quality assurance (QA) team evaluates how the various components of an application interact together in the full, integrated system or application.

System testing verifies that an application performs tasks as designed. This step, a kind of black box testing, focuses on the functionality of an application. System testing, for example, might check that every kind of user input produces the intended output across the application.

With system testing, a QA team gauges if an application meets all of its requirements, which includes technical, business and functional requirements. To accomplish this, the QA team might utilize a variety of test types, including performance, usability, load testing and functional tests.

With system testing, a QA team determines whether a test case corresponds to each of an application's most crucial requirements and user stories. These individual test cases determine the overall test coverage for an application, and help the team catch critical defects that hamper an application's core functionalities before release. A QA team can log and tabulate each defect per requirement.

Additionally, each individual type of system test reports relevant metrics of a piece of software, including:

- Performance testing: speed, average, stability and peak response times;
- Load testing: throughput, number of users, latency; and
- Usability testing: user error rates, task success rate, time to complete a task, user satisfaction.

System testing examines every component of an application to make sure that they work as a complete and unified whole. A QA team typically conducts system testing after it checks individual modules with functional or user-story testing and then each component through integration testing.

If a software build achieves the desired results in system testing, it gets a final check via acceptance testing before it goes to production, where users consume the software. An app-dev team logs all defects, and establishes what kinds and amount of defects are tolerable.

Various commercial and open source tools help QA teams perform and review the results of system testing. These tools can create, manage and automate tests or test cases, and they might also offer features beyond system testing, such as requirements management capabilities.

## 15.6 The Art of Debugging

In the context of software engineering, debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

It is a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected. Debugging is harder for complex systems in particular when various subsystems are tightly coupled as changes in one system or interface may cause bugs to emerge in another.

Debugging is a developer activity and effective debugging is very important before testing begins to increase the quality of the system. Debugging will not give confidence that the system meets its requirements completely but testing gives confidence.

**Definition:** Debugging is the process of detecting and removing of existing and potential errors (also called as ‘bugs’) in a software code that can cause it to behave unexpectedly or crash. To prevent incorrect operation of a software or system, debugging is used to find and resolve bugs or defects. When various subsystems or modules are tightly coupled, debugging becomes harder as any change in one module may cause more bugs to appear in another. Sometimes it takes more time to debug a program than to code it.

**Description:** To debug a program, user has to start with a problem, isolate the source code of the problem, and then fix it. A user of a program must know how to fix the problem as knowledge about problem analysis is expected. When the bug is fixed, then the software is ready to use. Debugging tools (called debuggers) are used to identify coding errors at various development stages. They are used to reproduce the conditions in which error has occurred, then examine the program state at that time and locate the cause. Programmers can trace the program execution step-by-step by evaluating the value of variables and stop the execution wherever required to get the value of variables or reset the program variables. Some programming language packages provide a debugger for checking the code for errors while it is being written at run time.

Here’s the debugging process

1. Reproduce the problem.

2. Describe the bug. Try to get as much input from the user to get the exact reason.

3. Capture the program snapshot when the bug appears. Try to get all the variable values and states of the program at that time.

4. Analyse the snapshot based on the state and action. Based on that try to find the cause of the bug.

5. Fix the existing bug, but also check that any new bug does not occur.

**Debugging Process:** Steps involved in debugging are:

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

**Debugging Strategies:**

1. Study the system for the larger duration in order to understand the system. It helps debugger to construct different representations of systems to be debugging depends on the need. Study of the system is also done actively to find recent changes made to the software.

2. Backwards analysis of the problem which involves tracing the program backward from the location of failure message in order to identify the region of faulty code. A detailed study of the region is conducting to find the cause of defects.

3. Forward analysis of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region

where the wrong outputs are obtained is the region that needs to be focused to find the defect.

4. Using the past experience of the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.

**Debugging Tools:**

Debugging tool is a computer program that is used to test and debug other programs. A lot of public domain software like gdb and dbx are available for debugging. They offer console-based command line interfaces. Examples of automated debugging tools include code based tracers, profilers, interpreters, etc.

Some of the widely used debuggers are:

- Radare2
- WinDbg
- Valgrind

**Difference between Debugging and Testing:**

Debugging is different from testing. Testing focuses on finding bugs, errors, etc whereas debugging starts after a bug has been identified in the software. Testing is used to ensure that the program is correct and it was supposed to do with a certain minimum success rate. Testing can be manual or automated. There are several different types of testing like unit testing, integration testing, alpha and beta testing, etc.

Debugging requires a lot of knowledge, skills, and expertise. It can be supported by some automated tools available but is more of a manual process as every bug is different and requires a different technique, unlike a pre-defined testing mechanism.

## 15.7 Approaches for Debugging

**1) Brute Force Method:** This method is most common and least efficient for isolating the cause of a software error. We apply this method when all else fail. In this method, a printout of all registers and relevant memory locations is obtained and studied. All dumps should be well documented and retained for possible use on subsequent problems.

**2) Back Tracking Method:** It is a quite popular approach of debugging which is used effectively in case of small applications. The process starts from the site where a particular symptom gets detected, from there on backward tracing is done across the entire source code till we are able to lay our hands on the site being the cause. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

**3) Cause Elimination:** The third approach to debugging, cause elimination, is manifested by induction or deduction and introduces the concept of binary partitioning. This approach is also called induction and deduction. Data related to the error occurrence are organized to isolate potential causes. A “cause hypothesis” is devised and the data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminated each. If initial tests indicate that a particular cause hypothesis shows promise, the data are refined in an attempt to isolate the bug.

**Tools for Debugging:** Each of the above debugging approaches can be supplemented with debugging tools. For debugging we can apply wide variety of debugging tools such as debugging compilers, dynamic debugging aids, automatic test case generators, memory dumps and cross reference maps. The following are the main Debugging tools available in the market.

**The following information is based upon details from respective web-sites of the tool vendors**

**1) Turbo Debugger for Windows:** The first debugger that comes to mind when you think of a tool especially suited to debug your Delphi code is of course Borland’s own turbo debugger for windows.

**2) HeapTrace:** Heap trace is a shareware heap debugger for Delphi 1-X and 2-X applications that enables debugging of heap memory use. It helps you to find memory leaks, dangling pointers and memory overruns in your programs. It also provides optional logging of all memory allocations, de-allocations and errors. Heap trace is optimized for speed, so there is only a small impact on performance even on larger applications. Heap trace is configurable and you can also change the format and destination of logging output, choose what to trace etc. You can trace each allocation and de-allocation and find out where each block of memory is being created and freed. Heap trace can be used to simulate out of memory condition to test your program in stress conditions.

**3) MemMonD:** MemMonD is another shareware memory monitor for Delphi 1-X applications. It is a stand-alone utility for monitoring memory and stack use. You don’t need to change your source code but only need to compile with debug information included,MemMonD detects memory leaks, de-allocations with wrong size.

**4) Re-Act:** Re-Act is not really a debugger for Delphi, but more a Delphi component Tester. However, this tool is a real pleasure to use. So we could not resist including it in this list of debugging tools. Currently, reach is for Delphi 2 only, but a 16 bits version is in the works. React version 2.0 is a really rice component, view and change properties at run time with the built in component inspector, monitor events in real time, set breakpoints and log events visually and dynamically. You can find elusive bugs, evaluate third party components and learn how poorly documented components really work. If you build or purchase components you need this tool. It’s totally integrated with Delphi 2.0 and the CDK 2.0.

### Check Your Progress

1. _________ is a testing based on user requirements and function processing.

2. Which of the following is a tool for acceptance testing?

(a) JTest (b) Watir (c) NTest (d) Jmeter

3. I am a black box testing technique to evaluate the the system’s compliance against specified requirements. Who Am I?

4. What is the other name for errors in software Engineering?

5. _____________ is a debugging approach that is most common and least efficient.

## 15.8 Software Maintenance

Software Maintenance is the process of modifying a software product after it has been delivered to the customer. The main purpose of software maintenance is to modify and update software application after delivery to correct faults and to improve performance.

**Need for Maintenance –**

Software Maintenance must be performed in order to:

- Correct faults.
- Improve the design.
- Implement enhancements.
- Interface with other systems.
- Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.
- Migrate legacy software.
- Retire software.

**Categories of Software Maintenance –**

Maintenance can be divided into the following:

1. **Corrective maintenance:**

   Corrective maintenance of a software product may be essential either to rectify some bugs observed while the system is in use, or to enhance the performance of the system.

2. **Adaptive maintenance:**

   This includes modifications and updations when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware and software.

3. **Perfective maintenance:**

   A software product needs maintenance to support the new features that the users want or to change different types of functionalities of the system according to the customer demands.

4. **Preventive maintenance:**

   This type of maintenance includes modifications and updations to prevent future problems of the software. It goals to attend problems, which are not significant at this moment but may cause serious issues in future.

**Reverse Engineering –**

Reverse Engineering is processes of extracting knowledge or design information from anything man-made and reproducing it based on extracted information. It is also called back Engineering.

**Software Reverse Engineering –**

Software Reverse Engineering is the process of recovering the design and the requirements specification of a product from an analysis of it's code. Reverse Engineering is becoming important, since several existing software products, lack proper documentation, are highly unstructured, or their structure has degraded through a series of maintenance efforts.

**Why Reverse Engineering?**

- Providing proper system documentatiuon.
- Recovery of lost information.
- Assisting with maintenance.
- Facility of software reuse.
- Discovering unexpected flaws or faults.

**Used of Software Reverse Engineering –**

- Software Reverse Engineering is used in software design, reverse engineering enables the developer or programmer to add new features to the existing software with or without knowing the source code.
- Reverse engineering is also useful in software testing, it helps the testers to study the virus and other malware code .

**Types of Software Maintenance**

There are four types of maintenance, namely, corrective, adaptive, perfective, and preventive. Corrective maintenance is concerned with fixing errors that are observed when the software is in use. Adaptive maintenance is concerned with the change in the software that takes place to make the software adaptable to new environment such as to run the software on a new operating system. Perfective maintenance is concerned with the change in the software that occurs while adding new functionalities in the software. Preventive maintenance involves implementing changes to prevent the occurrence of errors. The distribution of types of maintenance by type and by percentage of time consumed.

Corrective maintenance deals with the repair of faults or defects found in day-today system functions. A defect can result due to errors in software design, logic and coding. Design errors occur when changes made to the software are incorrect, incomplete, wrongly communicated, or the change request is misunderstood. Logical errors result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow, or incomplete test of data. All these errors, referred to as residual errors, prevent the software from conforming to its agreed specifications. Note that the need for corrective maintenance is usually initiated by bug reports drawn by the users.

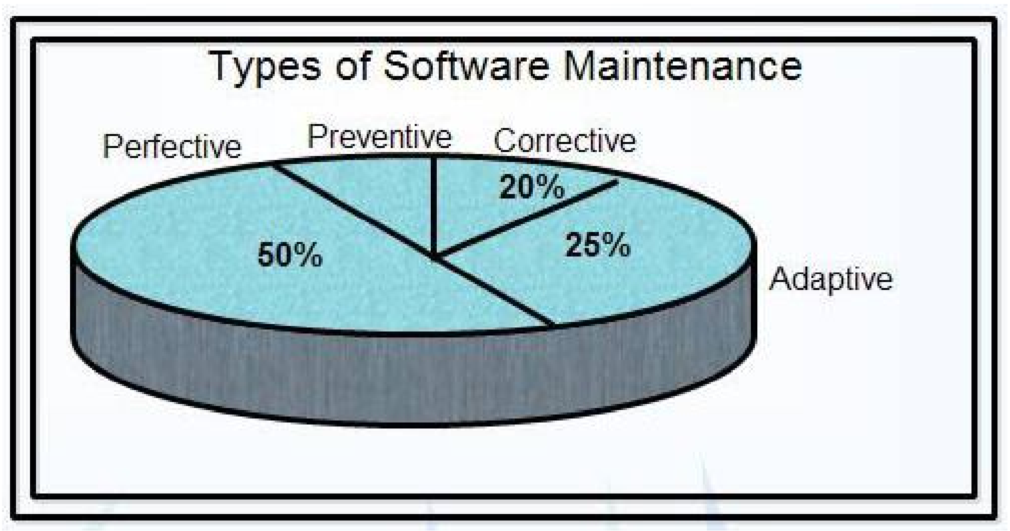


**Figure Types of Software Maintenance**

In the event of a system failure due to an error, actions are taken to restore the operation of the software system. The approach in corrective maintenance is to locate the original specifications in order to determine what the system was originally designed to do. However, due to pressure from management, the maintenance team sometimes resorts to emergency fixes known as patching. Corrective maintenance accounts for 20% of all the maintenance activities.

### Adaptive Maintenance

Adaptive maintenance is the implementation of changes in a part of the system, which has been affected by a change that occurred in some other part of the system. Adaptive maintenance consists of adapting software to changes in the environment such as the hardware or the operating system. The term environment in this context refers to the conditions and the influences which act (from outside) on the system. For example, business rules, work patterns, and government policies have a significant impact on the software system.

For instance, a government policy to use a single ‘European currency’ will have a significant effect on the software system. An acceptance of this change will require banks in various member countries to make significant changes in their software systems to accommodate this currency. Adaptive maintenance accounts for 25% of all the maintenance activities.

### Perfective Maintenance

Perfective maintenance mainly deals with implementing new or changed user requirements. Perfective maintenance involves making functional enhancements to the system in addition to the activities to increase the system’s performance even when the changes have not been suggested by faults. This includes enhancing both the function and efficiency of the code and changing the functionalities of the system as per the users’ changing needs.

Examples of perfective maintenance include modifying the payroll program to incorporate a new union settlement and adding a new report in the sales analysis system. Perfective maintenance accounts for 50%, that is, the largest of all the maintenance activities.

### Preventive Maintenance

Preventive maintenance involves performing activities to prevent the occurrence of errors. It tends to reduce the software complexity thereby improving program understandability and increasing software maintainability. It comprises documentation updating, code optimization, and code restructuring. Documentation updating involves modifying the documents affected by the changes in order to correspond to the present state of the system. Code optimization involves modifying the programs for faster execution or efficient use of storage space. Code restructuring involves transforming the program structure for reducing the complexity in source code and making it easier to understand.

Preventive maintenance is limited to the maintenance organization only and no external requests are acquired for this type of maintenance. Preventive maintenance accounts for only 5% of all the maintenance activities.

## 15.9 Summary

Software maintenance is a vast activity which includes optimization, error correction, and deletion of discarded features and enhancement of existing features. Since these changes are necessary, a mechanism must be created for estimation, controlling and making modifications. The essential part of software maintenance requires preparation of an accurate plan during the development cycle. Typically, maintenance takes up about 40-80% of the project cost, usually closer to the higher pole. Hence, a focus on maintenance definitely helps keep costs down.

## 15.10 Check Your Answers

1. Acceptance Testing.

2. (b) Watir

3. System Testing.

4. Bugs.

5. **Brute Force Method.**

## 15.11 Model Questions

1. What are the advantages of Validation?

2. Explain the steps to be performed in Acceptance Testing with a neat diagram.

3. What are the Advantages and Disadvantages of Acceptance Testing?

4. Define System Testing.

5. List out the different types of System Testing.

6. What is the strategy of Debugging?

7. Differentiate Debugging and Testing.

8. Explain about the Software maintenance in detail.

# MODEL QUESTION PAPER

## M.C.A – COMPUTER APPLICATION

## SECOND YEAR - FOURTH SEMESTER

## CORE PAPER - XIX

## SOFTWARE ENGINEERING

**TIME: 3 hrs** **Marks: 80**

**Part - A**

**I. Answer any TEN of the following in about fifty words each: (10 X2 = 20 marks)**

1. What is Software Engineering?
2. List out the advantages and disadvantages of waterfall Model.
3. What is Agile Modelling?
4. What are the different Data Models? Define them.
5. What is HIPO Diagram?
6. Write short notes on Taxonomy of Styles and Patterns.
7. Define Coupling and Cohesion.
8. What is Change Control?
9. Define Software Measurement.
10. Write short notes on Project Estimation Techniques.
11. What are the tasks of Risk Management?
12. What is the need for software review?

## Part - B

**II. Write Short Notes on any SIX of the following, in about 250 words each:**

**(6 X 5 = 30 marks)**

13. Write in brief about the various type of software.
14. Explain in brief about phases of Adaptive Software Development.
15. Describe about different types of checks to be carried out during the requirements validation process.
16. Write briefly about different approaches used in agile methods
17. Explain in brief about components of Data Flow Diagram.
18. Write in brief about Project Estimation Techniques.
19. What are the methods of Risk Identification? Write short notes on them.
20. Describe in brief about the difference between validation and verification.

## Part - C

**III. Write Essay on any THREE of the following, in about 750 words each.**

**(3 x 10 = 30 marks)**

21. Explain in detail about Software Process Models.
22. Describe in detail about the elements of Flow Oriented Model.
23. Illustrate the difference between coupling and cohesion? Explain about Coupling and Cohesion.
24. Elaborate about W5HH Principle.