GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

SOFTWARE ENGINEERING

Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth.

Program Educational Objectives

- Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.
- Manage software projects with significant technical, legal, ethical, social, environmental and economical considerations.
- Demonstrate commitment and progress in lifelong learning, professional development, and leadership and communicate effectively with professional clients and the public.

HANDOUT ON SOFTWARE ENGINEERING

Class& Sem. : III B.Tech – II Semester Year : 2018-19

Branch : CSE Credits : 3

==

1. Brief History and Scope of the Subject

Software engineering is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind. This course covers the fundamentals of software engineering, including understanding system requirements, finding appropriate engineering compromises, effective methods of design, coding, and testing, team software development, and the application of engineering tools. The course will combine a strong technical focus with a capstone project providing the opportunity to practice engineering knowledge, skills, and practices in a realistic development setting with a real client.

2. Pre-Requisites

- Familiar with the fundamental concepts of computers.

3. Course Objectives:

- Illustrate basic taxonomy and terminology of the software engineering.
- Plan and monitor the control aspects of project.

4. Course Outcomes:

Upon successful completion of the course, the students will be able to

CO1: explain the basic concepts of Software Engineering.

CO2: select the suitable process model based on the client requirements.

CO3: calculate software proficiency in terms of cost and schedule.

CO4: list the specifications of end-user according to business needs.

CO5: choose the appropriate architectural style for a given Scenario.

CO6: infer the system model for a sample case study.

CO7: deduce test cases by following different testing methodologies.

CO8: Explore the basic concepts of software engineering.

5. Program Outcomes:

Graduates of the Computer Science and Engineering Program will have

a) an ability to apply knowledge of mathematics, science, and engineering

b) an ability to design and conduct experiments, as well as to analyze and interpret data

c) an ability to design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability

d) an ability to function on multidisciplinary teams

e) an ability to identify, formulate, and solve engineering problems

f) an understanding of professional and ethical responsibility

g) an ability to communicate effectively

h) the broad education necessary to understand the impact of engineering solutions in a global, economic, environmental, and societal context

i) a recognition of the need for, and an ability to engage in life-long learning,

j) a knowledge of contemporary issues

k) an ability to use the techniques, skills, and modern engineering tools necessary for engineering practice.

6. Mapping of Course Outcomes with Program Outcomes:

	a	B	c	d	e	f	G	h	i	j	k
CO1							**H**				
CO2		**M**									
CO3					**M**						
CO4		**M**									
CO5										**L**	
CO6											
CO7											
CO8											

7. Prescribed Text Books

a. Pankaj Jalote, “A Concise Introduction to Software Engineering”, Springer International Edition.

b. Roger S. Pressman, “Software Engineering”, 7th edition, TMH.

8. Reference Text Books

a. K.K Aggarwal and Yogesh Singh, “Software Engineering”, 3rd Edition, New Age Publications.

b. Sommerville, “Software Engineering”, 8th edition, Pearson.

9. URLs and Other E-Learning Resources

a. https://www.learningware.in

b. http://www.learnerstv.com/engineering.php

c. http:/www.mhhe.com/pressman

d. http:/www.software-engin.com

e. http:/www.sei.cmu.edu

f. http:/www.scitools.com

g. http:/www.galorath.com

10. Digital Learning Materials:

- https://onlinecourses.nptel.ac.in

11. Lecture Schedule / Lesson Plan

Topic	**No. of Periods**	
	Theory	**Tutorial**
UNIT –1: Introduction to Software Engineering		
The evolving role of software	1	1
Changing nature of software	2	
Software myths	2	
The software problem: cost, schedule and quality	2	1
Scale and change	1	
	8	**2**
UNIT – 2: Software Process		
Process and project	1	1
Software development process models: waterfall model	2	
Prototyping , Iterative development	2	1
Relational unified process, Extreme programming and agile process.	3	
	8	**2**
UNIT – 3: Planning a software project		
Effort estimation	2	1
Project schedule and staffing	2	
Quality planning	2	1
risk management planning	2	
	8	**2**

UNIT – 4: Software requirement analysis and specification		
Introduction, Value of good SRS	2	1
Requirement process, Requirement specification	3	1
functional specification with use cases	3	
	8	**2**
UNIT – 5: Software Architecture and Design		
Role of software architecture, architecture views	2	2
Components and connector view, architecture styles for C & C view	3	
Function-oriented design	2	
Object oriented design	2	1
Metrics for design	2	
	11	**3**
UNIT – 6: Coding and Unit testing		
Programming principles and guidelines	2	1
Testing concepts, testing process	2	
Black-box testing, white-box testing	3	2
Metrics for testing	2	
	9	**3**
Total No. of Periods:	**52**	**14**

12. Seminar Topics

- Eye Tracking Software
- Agile Supply Chain
- Reconfigurable Manufacturing System
- Micro Air Vehicle
- Adhoc Wireless Networks
- Software Testing
- Liquid Lens
- Monorail
- Artificial Eye
- Biometric Voting System
- Infrared Plastic Solar Cell
- Solar Mobile Charger

Unit – I

INTRODUCTION TO SOFTWARE ENGINEERING

1. The Evolving role of software

- The role of computer software has undergone significant change over the last 50 years.
- Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have lead to the development of sophisticated and complex computer-based systems.
- Popular books published during the 1970s and 1980s described the changing nature of computers and software and their impact on our culture.
- Some of them stated that computers and software
    - Caused new industrial revolution
    - Lead to a transformation from an industrial society to an information society
    - Are the key to knowledge interchange throughout the world
- As the 1990s began, computers and software lead to a democratization of knowledge.
- During the later 1990s, the internet became very popular and lead to the development of web-based software systems. During this time many sectors like banking, insurance, airlines etc. have automated.
- Today, ubiquitous computing has created a generation of information appliances that have connectivity to the Web to provide a blanket of connectedness over our homes, offices and motorways.
- Software’s role continues to expand still.

1.2 Software

- Software is a set of

  Instructions that when executed provide desired function and performance,

Data Structures that enable the programs to manipulate information, **Documents** that describe the operation and use of the programs.

- Software is a logical rather than a physical system element.

1.2.1 Characteristics of Software

Every software exhibits three kids of characteristics.

1. *Software is developed or engineered, it is not manufactured*
    - Hardware is manufactured, but software is developed.
    - Both activities require the construction of a product but the approaches are different.
2. *Software doesn't "wear out"*
    - Environmental problems such as dust, vibration, abuse, temperature extremes etc. may cause the hardware to wear out. On the other hand, environmental problems can't influence the software and therefore it does not wear out.
    - Figure 1.1 depicts failure rate as a function of time for hardware. The relationship is often called the "bathtub curve". It indicates that hardware exhibits relatively high failure rates early in its life.
    - These failures are often due to design or manufacturing defects.
    - These defects can be corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time.
    - As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies, which means that the hardware begins to wear out.

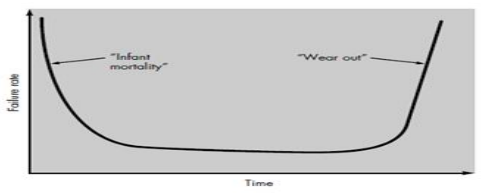


Figure 1.1

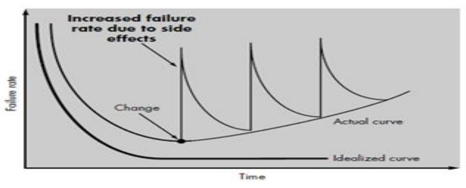


Figure 1.2

- Figure 1.2 depicts failure rate curve of software and it takes the form of "idealized curve".

- Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown in Figure 1.2.
- During its life, software will undergo change (maintenance).
- As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.2.

3. *Although the industry is moving toward component-based assembly, most software continues to be custom built*
   - A software component should be designed and implemented so that it can be reused in many different programs.
   - Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts.
   - For example, today's graphical user interfaces are built using reusable components such as windows, pull-down menus, and a wide variety of interaction mechanisms.

2. Changing nature of software

- The nature of software has changed a lot over the years. It has changed from writing programs by individuals for their personal use to writing very complex software to run a nuclear power plant.
- Its nature mainly depends on the type of software used. Given below are the different types of software being used in different applications.

System Software

- System Software is a collection of programs written to provide service to other programs.
- It needs heavy interaction with computer hardware.
- It contains complex data structures and multiple external interfaces

  Examples: Compilers, Editors, File Management Utilities, other System Applications Drivers and Networking Software.

Application Software

- Application Software consists of standalone programs that are used to solve specific business needs.
- It is used to process technical data/technical decisions and control business functions in real time.

  Examples: Conventional Data Processing Applications, Real-Time Manufacturing Process Control, point-of-sale etc.

Engineering/Scientific Software

- It is characterized by conventional numerical algorithms.
- It is used to create interactive applications to take on real time.

  Examples: Computer Aided Design(CAD/CAM), System Simulation, Weather prediction system, Interactive Applications in Educational Field.

Embedded Software

- Software that resides within a product or system is called as Embedded Software.

  Examples: Keypad control for a Microwave Oven, Smart dustbins etc.

Product-line Software

- This type of software provides specific capability for use by many different customers.

  Examples: Word Processing, Spreadsheets, Computer Graphics, Database Management, Multimedia & Entertainment and Business Financial Applications.

Web Applications

- It can be considered as a set of linked hypertext files.
- Web Application Software has grown relevant as E-Commerce & B2B applications grow in importance.

  Examples: E-commerce sites, Air line reservation system, IRCTC etc.

Artificial Intelligence Software

- This type of software uses Non-Numerical Algorithms to solve complex problems.

Examples: Robotics, Expert Systems, Pattern Recognition(image and voice), Artificial Neural Networks, Theorem Proving, Game Playing.

Open Source

- Open Source Software refers to the software whose source code is public to everyone to develop, test or improve.

  Examples: Linux Operating System, Apache Web Server Application, LibreOffice Application, GNU Image Manipulation Application.

3. Software myths

- Are the false beliefs that managers, customers, and developers have on the software development.

3.1 Management myths

Myth1: Development problems can be solved by developing and documenting standards.

Reality: Standards have been developed by companies and standards organizations. They can be very useful. However, they are frequently ignored by developers because they think that they are irrelevant and sometimes incomprehensible.

Myth2: Development problems can be solved by using state-of-the art tools.

Reality: Tools may help, but there is no magic. Problem solving requires more than tools. It requires great understanding.

Myth3: If we fall behind schedule in developing software, we can just put more people on it.

Reality: If software is late, adding more people will merely make the problem worse. This is because the people already working on the project need to educate the newcomers. So, this does not immediately reduce the work.

Myth4: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

3.2 Customer myths

Customers often underestimate the difficulty of developing software. Sometimes marketing people encourage customers in their misbeliefs.

Myth1: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

Reality: A poor definition of the problem is the major cause of failed software. A detailed description of the information domain, functions, behavior, performance, interfaces, design constraints, and validation criteria is essential before writing the programs. These can be determined only after thorough communication between customer and developer.

Myth2: Project requirements continuously change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. If change is requested in design, then it costs 5 times more as if it is done in analysis. If it is requested in coding, then it costs 10 more, in testing it is 50 times more and if it is requested after delivery, then its cost increases enormously.

3.3 Developer’s (Practitioner's) myths

Practitioner's often want to be artists, but the software development craft is becoming an engineering discipline. However myths remain:

Myth1: Once we write the program and get it to work, our job is done.

Reality: 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 60% of the development costs. This myth is true only for shelfware --- software that is never used.

Myth2: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms is formal technical review. Software reviews can effectively detect the problems in requirements documents, design documents, test plans, and code.

Myth3: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software delivery. Apart from this several other documents such as analysis, design and testing documents, user manuals etc. may also be created during software development.

Myth4: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

4. The Software Problem

- The software developed by a student as part of his laboratory work consists of few hundred Lines of Code (LOC). Such systems tend to have a very limited purpose and a very short life span. We can afford to throw them away and replace them with an entirely new software rather than attempt to reuse them or repair them.
- On the other hand, industrial strength software is developed by a team of people and consists of few thousand Lines of Code (LOC). These are the applications that exhibit a very rich set of behaviors and are used by a large number of customers. Such systems tend to have a very long life span. Ex. Air-traffic control system, Railway Reservation System etc.
- Thus, the problem domain for software engineering is developing an industrial strength software. This should be produced at reasonable cost, in a reasonable time, and should be of good quality.

- Thus, the basic elements of industrial strength software are cost, schedule and quality.

4.1 Cost

- Cost of software is measured based on its size. The size of the software is measured in Lines of Code (LOC) or Thousand Lines of Code (KLOC).
- As the main cost of producing software is the manpower employed, the cost of developing software measured in terms of LOC (or) KLOC per person-months.
- Generally, software companies charge the client between \$3000 - \$15000 per person month.
- For Example, assume that size of software is 50 million LOC and a person can write 5000 LOC in a month. Let the company charging \$6000 per person month.
- Then the cost of software can be calculated as

  Person months required = 500,00,000/5000 = 10000

  Cost = 10000X6000 = \$6,00,00,000

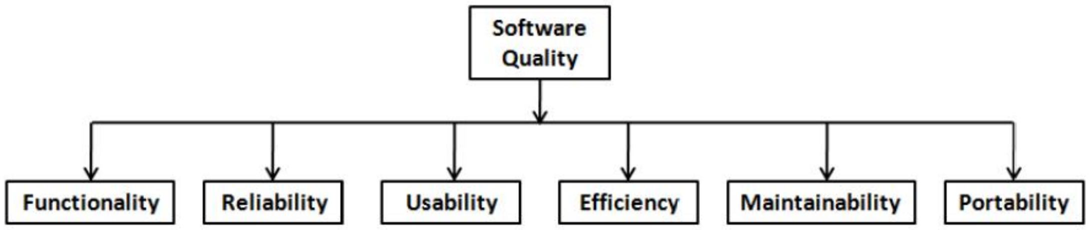
Schedule

- Schedule is very important for developing projects.
- Business trends are dictating that the time to market a product should be reduced. It means that software needs to be developed faster, and within the specified time.
- Unfortunately, the history of software shows that it is usually delivered late.
- Therefore, reducing the cost and time for software development are central goals of software engineering.

Quality

- Besides cost and schedule, the other major factor driving software engineering is quality.
- Today, quality is one of the main mantras, and business strategies are designed around it.

- The international standard on software quality suggests six main attributes of software quality as shown in the figure below.



Software quality attributes

These attributes can be defined as follows

- **Functionality:** The capability to provide functions which meet stated needs of software.
- **Reliability:** The capability to provide failure-free service.
- **Usability:** The capability to be understood, learned, and used.
- **Efficiency:** The capability to provide appropriate performance relative to the amount of resources used.
- **Maintainability:** The capability to be modified for purposes of making corrections, improvements, or adaptation.
- **Portability:** The capability to be adapted for different specified environments without applying actions.
- One measure of quality is the number of defects in the delivered software per unit size (generally taken to be thousands of lines of code, or KLOC).
- Current best practices in software engineering have been able to reduce the defect density to less than 1 defect per KLOC.

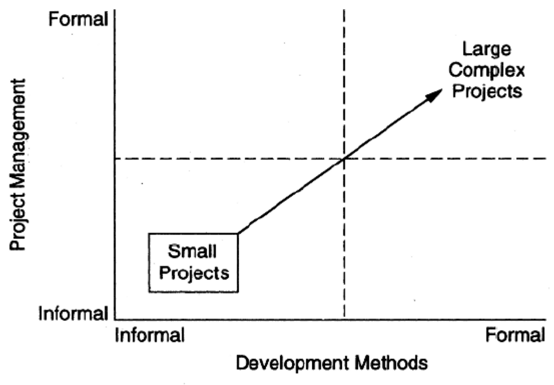
Though cost, schedule, and quality are the main driving forces of industry strength software, there are some other characteristics of it.

Scale

- Most industrial-strength software systems tend to be large and complex, requiring tens of thousands of lines of code. For example, sizes of some of the well-known software products are given below;

Software	Size (KLOC)
Windows 10	50,000
Google Chrome	7,000
Facebook	62,000
Ms Office	45,000
Google	2000,000
Human Genome	3300,000
Any Commercial Software (IRCTC, Banking s/w etc.)	50,000

- Developing a large system requires a different set of methods compared to developing a small system. i.e. the methods used for developing small systems often do not scale up to large systems.
- As an example, consider the problem of obtaining the opinion poll of people in a room as well as across the country. It is obvious that, the methods used for obtaining the opinion poll of people in a room will just not work for obtaining the opinion poll of people across the country. A different set of methods will have to be used for this and will require considerable management, execution and validation.
- Similarly, methods that one can use to develop programs of a few hundred lines of code cannot work for the software consisting of hundred thousand lines of code. A different set of methods must be used for developing such large software.
- Any software project involves the use of engineering and project management. In small projects, informal methods for development and management can be used. For large projects, more formal methods are used. This is shown below.



- As shown in the above figure, more formal methods are used for developing large and complex projects to make sure that cost, schedule, and quality are under control.

Change

- Change is another characteristic of the problem domain and should be handled properly.
- It is obvious that, all the requirements of the system are not known at the beginning. As the development proceeds and time passes, additional requirements are identified, which need to be incorporated in the software being developed. This requires that, suitable methods are to be developed to accommodate the change efficiently. Otherwise, change requests can trouble the project and can consume up to 30 to 40% of the development cost.

UNIT-I
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) What is Software? [ ]

a) Software is set of programs.

b) Software is documentation and configuration of data.

c) Both a and b

d) None of the mentioned

2) What are the characteristics of software? [ ]

a) Software is developed or engineered; it is not manufactured in the classical sense.

b) Software doesn't "wear out".

c) Software can be custom built or custom build.

d) All mentioned above

3) The process of developing a software product using software engineering principles and methods is referred to as, ____________. [ ]

a) Software myths

b) Scientific Product

c) Software Evolution

d) None of the above

4) Software consists of _________. [ ]

a) Set of instructions + operating procedures

b) Programs + documentation + operating procedures

c) Programs + hardware manuals

d) Set of program

5) The extent to which the software can continue to operate correctly despite the introduction of invalid inputs is called as [ ]

a) Reliability

b) Robustness

c) Fault Tolerance

d) Portability

e) All of the above.

6) As per an IBM report, “31%of the project get cancelled before they are completed, 53% overrun their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts”. What is the reason for these statistics? [ ]

a)Lack of adequate training in software engineering

b)Lack of software ethics and understanding

c)Management issues in the company

d) All of the mentioned

7) Compilers, Editors software comes under which type of software? [ ]

a) System software b) Application software

c) Scientific software d) None of the above

8) Which of the following cannot be applied with the software according to Software Engineering Layers? [ ]

a) Process b) Methods

c) Manufacturing d) None of the above.

9) Choose the correct option according to the given statement. [ ]

Statement 1: Software is a physical rather than a logical system element. Statement 2: Computer software is the product that software engineers design and build.

Statement 3: Software is a logical rather than a physical system element. Statement 4: Software is a set of application programs that are built by software engineers.

a) Statement 1 and 2 are correct.

b) Only Statements 2 and 3 are correct.

c) Statements 2, 3 and 4 are correct

10)From the following which quality deals with maintaining the quality of the software product? [ ]

a) Quality assurance b)Quality control

b) Quality efficiency d)None of the above

11)Which one of the following is not a symptom of the present software crisis: [ ]

a) Software is expensive

b) It takes too long to build a software product

c) Software is delivered late

d) Software products are required to perform very complex tasks

12) Which one of the following characteristics of software products being developed is not a symptom of software crisis? [ ]

a) Fail to meet user requirements b) Expensive

c) Highly interactive d) Difficult to alter, debug, and enhance

13) Why is writing easily modifiable code important? [ ]

a) Easily modifiable code results in quicker run time

b) Most real world programs require change at some point of time or other

c) Most text editors make it mandatory to write modifiable code

d) Several developers may write different parts of a large program

SECTION-B

SUBJECTIVE QUESTIONS

1) Define Software and Software Engineering? List out the important characteristics of software.
2) Discuss the changing nature of the software.
3) Identify different Myths and Realities related to software. Explain briefly.
4) Describe the major driving forces of a Software Project.
5) Illustrate different Software Quality Attributes? Explain briefly.
6) Give a conclusion about the statement “Software is easy to change, because Software is flexible”
7) Analyze how the Failure Curve of Hardware and Software can be differentiated?
8) Categorize some problems that will come up if the methods you currently use for developing small software are used for developing large software systems.
9) Suppose a program for solving a problem cost C and industrial strength software for solving that problem costs 10 C. where do you think this extra 9 C cost is spent? suggest a possible breakdown of this extra cost.

SECTION-C

GATE QUESTIONS

1) If you are given extra time to improve the reliability of the final product developing a software product, where would you spend this extra time?

Unit – II

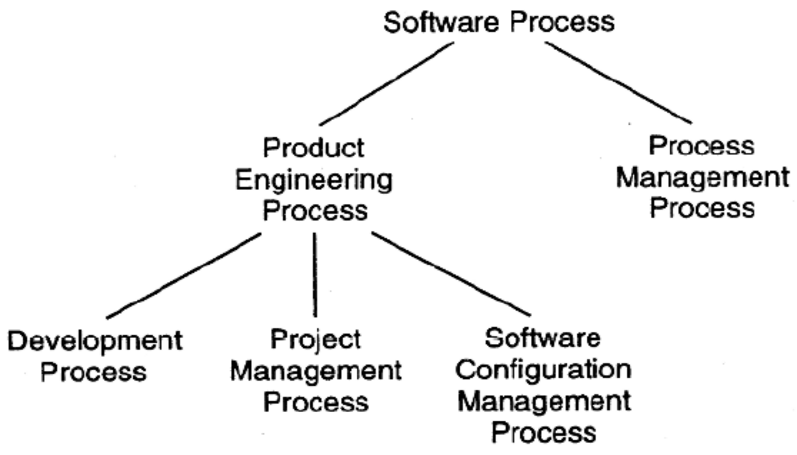
SOFTWARE PROCESS

1. Process and Project

- Software process – is a sequence of activities to be performed to make a software product.
- Several process models have been developed so far for the efficient development of software that meets the user requirements.
- Software process
  - Helps development of software in a systematic and disciplined manner.
  - Helps common understanding of activities among the software developers.
- Software project – is one instance of the software process.

1.2 Component Software Processes

- As defined above, software process is a sequence of steps executed to develop a software product.
- Several component processes exists in a software process. The relationship between the major component processes is shown below as a hierarchy;


Software Process
Product Engineering Process
Process Management Process
Development Process
Project Management Process
Software Configuration Management Process

- At the bottom, we have development process and project management process.

- The development process specifies all the engineering activities (analysis, design, coding, testing and maintenance) that need to be performed to develop a software product, whereas the management process specifies how to plan and control these activities so that cost, schedule, quality, and other objectives are met.
- During the project, many products are produced which are typically composed of many items (for example, the final source code may be composed of many source files).
- These items keep evolving as the project proceeds, creating many versions on the way.
- So, it is the responsibility of software configuration management process to handle these different versions.
- These three component processes viz. development process, project management process, software configuration management process comes under product engineering process, as their main objective is to produce the desired product.
- The basic objective of the process management process is to improve the software process. By improvement, we mean that the capability of the process to produce quality goods at low cost is improved.

2 Software Development Process Models

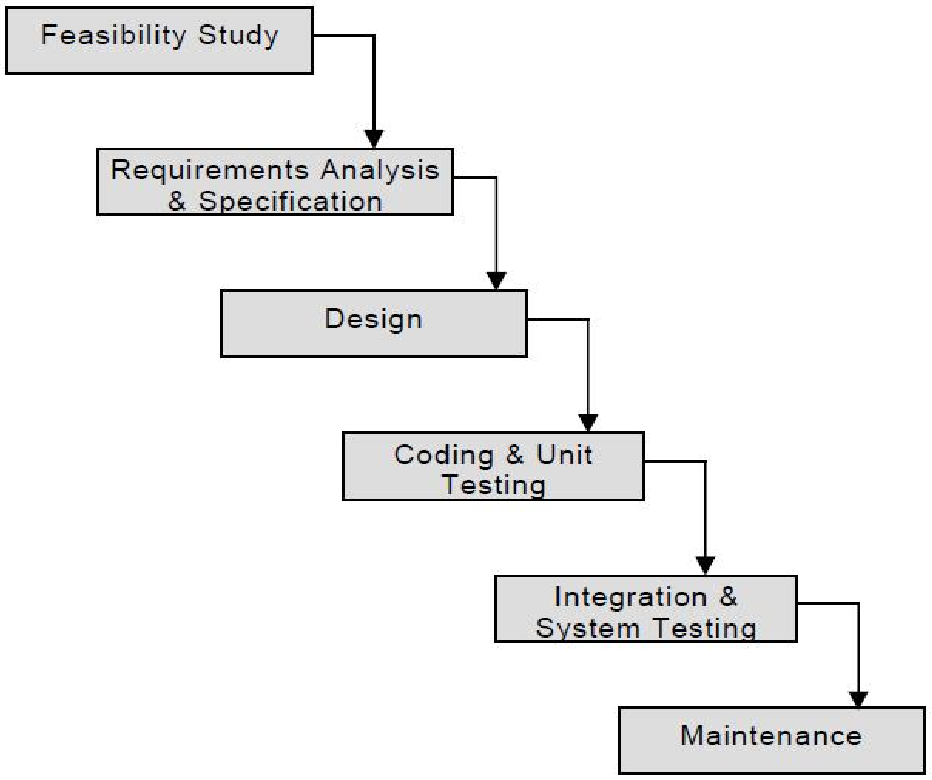
- Software development process defines all the activities undertaken during product development and establishes a precedence ordering among the different activities.
- The development team must identify a suitable process model and then adhere to it.
- The primary advantage of adhering to a life cycle model is that it helps development of software in a systematic and disciplined manner.
- Due to the importance of the development process, various models have been proposed.

2.1 Waterfall Model

- The simplest process model is the waterfall model, which states that the phases are organized in a linear order.
- Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects.
- But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.
- Classical waterfall model divides the life cycle into the following phases as shown in the following figure.

Feasibility Study

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


Feasibility Study
Requirements Analysis & Specification
Design
Coding & Unit Testing
Integration & System Testing
Maintenance

Classical Waterfall Model

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.
- After they have an overall understanding of the problem they investigate different possible solutions. Then they examine each of these solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

The following is an example of a feasibility study undertaken by an organization. It is intended to give you a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Case Study

A mining company named Galaxy Mining Company Ltd. (GMC) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of mines at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident fund (SPF) would be quickly distribute some compensation before the standard provident amount is paid. According to this scheme, each mine site would deduct SPF installments from each miner every month and deposit the same with the CSPFC (Central Special Provident Fund Commissioner). The CSPFC will maintain all details regarding the SPF installments collected from the miners. GMC employed a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. GMC realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. GMC indicated that the amount it can afford for this software to be developed and installed is Rs. 50 millions.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed the matter with the top managers of GMC to get an overview of the project. He also discussed the issues involved with the several field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One was to have a central database which could be accessed and updated via a satellite connection to various mine sites. The other approach was to have local databases at each mine site and to update the central database periodically through a dial-up connection. These periodic updates could be done on a daily or hourly basis depending on the delay acceptable to GMC in invoking various functions of the software. The project manager found that the second approach was very affordable and more fault-tolerant as the local mine sites could still operate even when the communication link to the central database temporarily failed. The project manager quickly analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

Requirements Analysis and Specification

- The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely
  - Requirements gathering and analysis, and
  - Requirements specification
- The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.
- The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions.
- For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements.
- The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system.
- Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer.
- After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.

- The important components of this document are functional requirements, the non-functional requirements, and the goals of implementation.

Design

- The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
- In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.
- **Traditional design approach -** Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.
- **Object-oriented design approach -** In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and Unit Testing

- The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.
- During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module

in isolation as this is the most efficient way to debug the errors identified at this stage.

Integration and System Testing

- Integration of different modules is undertaken once they have been coded and unit tested.
- During the integration and system testing phase, the modules are integrated in a planned manner.
- The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps.
- During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it.
- Finally, when all the modules have been successfully integrated and tested, system testing is carried out.
- The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:
    - **α – testing:** It is the system testing performed by the development team.
    - **β – testing:** It is the system testing performed by a friendly set of customers.
    - **Acceptance testing:** It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.
- System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

Maintenance

- Maintenance of a typical software product requires much more than the effort necessary to develop the product itself.
- Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio.
- Maintenance involves performing any one or more of the following three kinds of activities:
  - **Corrective maintenance** - Correcting errors that were not discovered during the product development phase.
  - **Perfective maintenance** - Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements.
  - **Adaptive maintenance** - Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Limitations of Waterfall Model

- The waterfall model, although widely used, has some strong limitations. Some of the key limitations are:
  1. It assumes that the requirements of a system can be frozen (i.e., baselined) before the design begins. This is possible for small systems. But for large systems, determining the requirements is difficult as the user does not even know the requirements.
  2. Freezing the requirements usually requires choosing the hardware. A large project might take few years to complete. If the hardware is selected early, then due to the speed at which hardware technology is changing, it is likely that the final software will use a hardware that becomes obsolete. This is clearly not desirable for such expensive software systems.

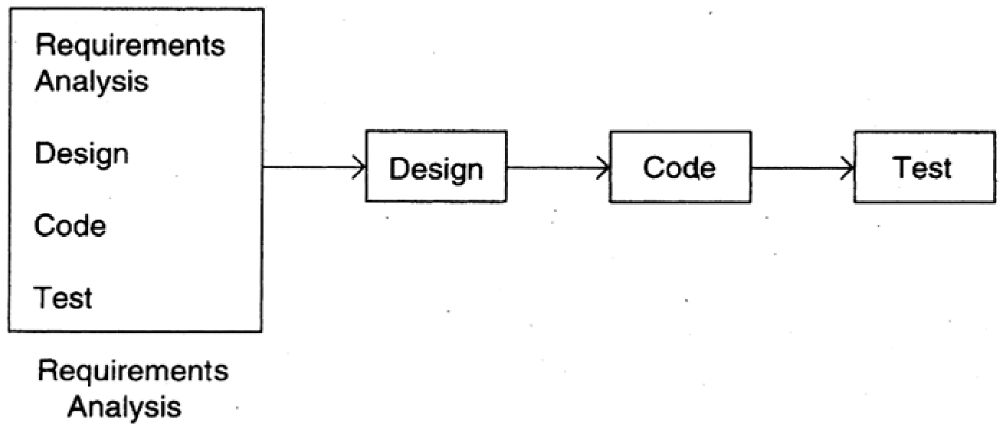
3. It follows the "big bang" approach—the entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know until the end what they are getting. Furthermore, if the project runs out of money in the middle, then there will be no software. That is, it has the "all or nothing" value proposition.
4. It encourages "requirements bloating". Since all requirements must be specified at the start, it encourages the users and other stakeholders to add unnecessary requirements (which they think that they are needed but, may not be used finally).
5. It is a document-driven process that requires formal documents at the end of each phase.

When to use the Waterfall Model?

- Requirements are well known and stable
- Technology is understood
- Development team have experience with similar projects
- For small projects

2.2 Prototyping

- The goal of a prototyping-based development process is to counter the first limitation of the waterfall model.
- The basic idea here is that instead of freezing the requirements before any design or coding can proceed, a throwaway prototype is built to help understand the requirements.
- A prototype is a toy implementation of the system. It usually exhibits limited functional capabilities.
- This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is done more informally.
- By using this prototype, the client can get an actual feel of the system, which can enable the client to better understand the requirements of the desired system. This results in more stable requirements.
- The process model of the prototyping approach is shown below.



- The development of the prototype proceeds as follows; with currently known requirements, a prototype of the system is developed. The prototype is then presented to the clients/users. Based on the feedback given by the clients/users, the prototype is modified to incorporate the suggested changes. This process is continued until the clients/users are satisfied with the prototype.
- Once the prototype is accepted, it means that all the requirements of the system have obtained. After this, the actual design, coding, and testing phases are carried out to develop the software.

Advantages

- Provides a working model to the user early in the process, so, it increases user's confidence.
- The developer gains experience and insight by developing a prototype that results better implementation of requirements.
- The prototyping model serves to clarify requirements which are not clear; hence, it reduces ambiguity and improves communication between the developers and users.
- There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent.
- Helps in reducing risks associated with the software.

Disadvantages

- If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive.
- The developer loses focus of the real purpose of prototype and hence, may compromise with the quality of the software. For example, developers may use some inefficient algorithms or inappropriate programming languages while developing the prototype.
- Prototyping can lead to false expectations. For example, a situation may be created where the user believes that the development of the system is finished when it is not.

Where to Use Prototype Model?

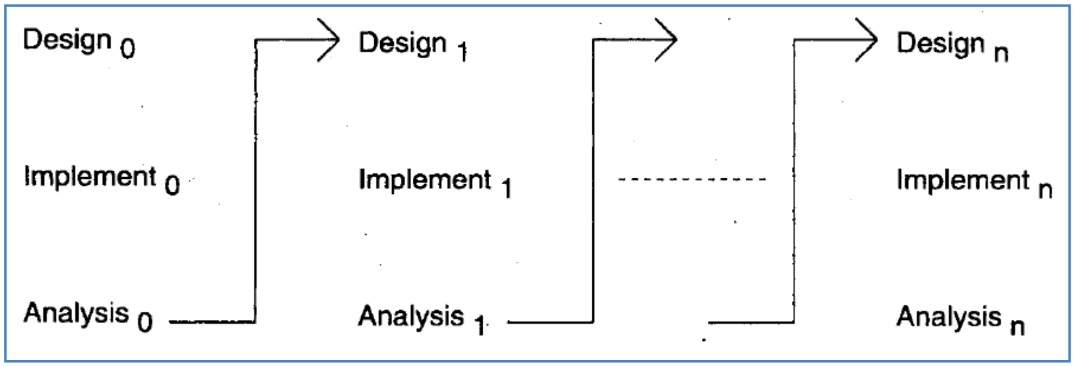
- Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements.
- This might be needed for novel systems.

2.3 Iterative Development

- The iterative development process model counters the third and fourth limitations of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model.
- The basic idea is that the software should be developed in increments, each increment adding some functionality to the system until the full system is implemented.
- There are two approaches in iterative model.
- The first approach called the *iterative enhancement model* works as follows;
- A simple initial implementation is done for a subset of requirements that are assumed to be important for the customer.
- A *project control list* is created that contains, in order, all the tasks that must be performed to obtain the final implementation.
- Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the

implementation, and performing an analysis of the partial system obtained. These three phases are called the design phase, implementation phase, and analysis phase.

- The process is iterated until the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement model is shown below.



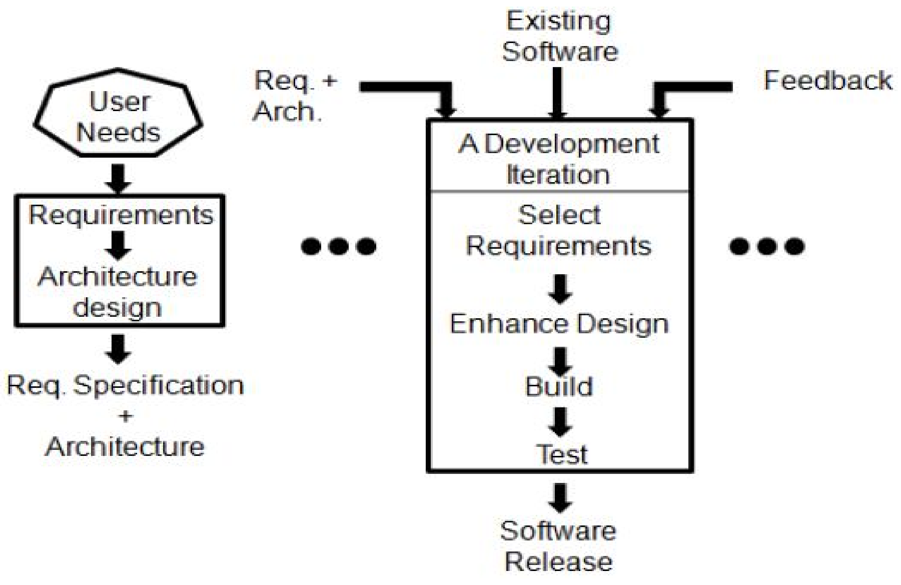
Iterative Enhancement Model

- The second approach for iterative development is to do the requirements and the architecture design in a standard waterfall or prototyping approach, but deliver the software iteratively.

- At each iteration which requirements will be implemented in this release is decided, and then the design is enhanced and code is developed to implement the requirements. The iteration ends with delivery of a working software system.

- This process is repeated until the final system is released.

- Selection of requirements for an iteration is done primarily based on the value the requirement provides to the end users. This approach is shown below.



Iterative delivery approach

Advantages

- In today's world clients do not want to invest too much money without seeing returns. The iterative model permits this—after each iteration some working software is delivered.
- It reduces the rework.
- It has no all-or-nothing risk.
- New requirements can be added easily.

Disadvantages

- Needs good planning and design.
- Becomes invalid when there is time constraint on the project schedule.

2.4 Rational Unified Process

- Rational Unified Process (RUP) is another iterative process model that was designed by Rational Software, now part of IBM.
- It was primarily designed for object-oriented development.

- In this, software development is divided into cycles where each cycle consists of four phases;
    - Inception phase
    - Elaboration phase
    - Construction phase
    - Transition phase

Inception Phase

- Inception phase is first phase of the process
- The objectives of the inception phase are;
    - Establishing the project's software scope and boundary conditions including a clear understanding of what is and is not intended in the product.
    - Illustrating the critical use cases of the system.
    - Demonstrating at least one candidate architecture.
    - Estimating the cost and schedule for the entire project.
    - Estimating the potential risk.

Elaboration Phase

- Elaboration is the second phase of the process
- The objectives of the Elaboration phase are;
    - Baselining the architecture.
    - Baselining the vision.
    - Baselining a plan for the construction phase.
    - Demonstrating that the architecture will support the vision at a reasonable cost in a reasonable time.

Construction Phase

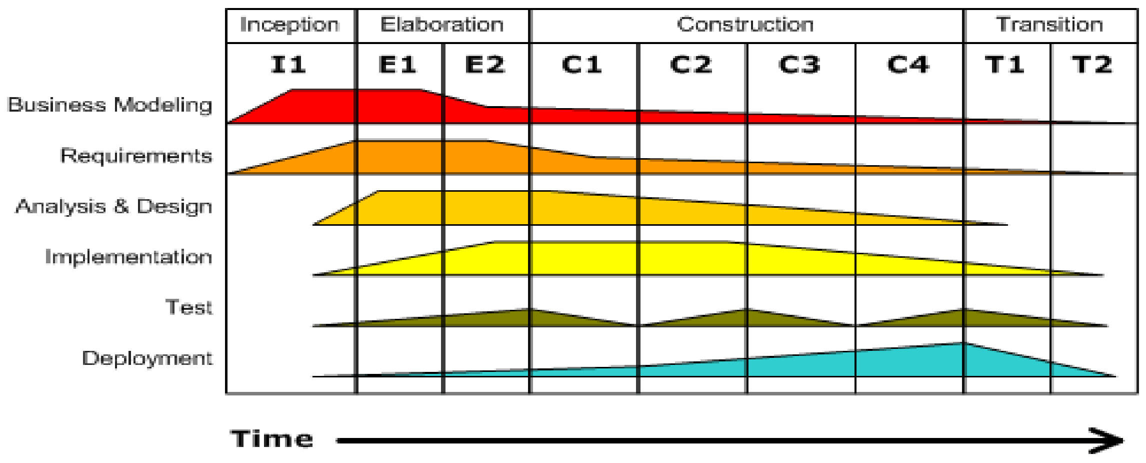
- Construction is the third phase of the process which involves the construction of the software system.

- The objectives of the Construction phase are;
    - Minimizing the development costs by optimizing the resources and avoiding the unnecessary scrap and rework.
    - Achieving adequate quality.
    - Achieving useful versions (alpha, beta and other test releases).

Transition Phase

- Transition is the fourth phase of the process which involves the deployment of the software system to the user community.
- The objectives of the Transition phase are;
    - Beta testing to validate the new system against user expectations.
    - Installing the software at the client sites.
    - Conversion of operational databases.
    - Training of users and maintainers.

The RUP model is shown below.



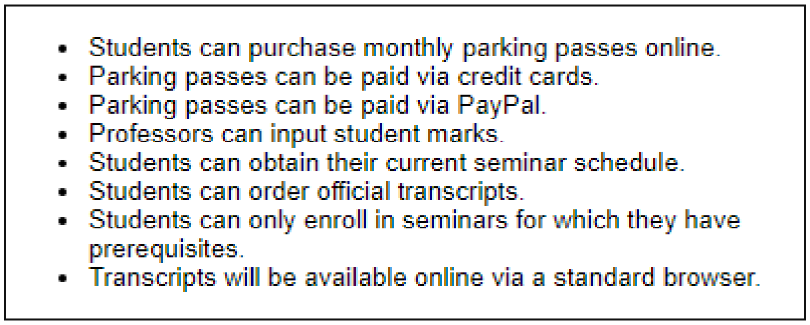
2.5 Extreme Programming and Agile Process

- To overcome the shortcomings of the waterfall model of, Agile process model was proposed in mid 1990s.

- The agile model was primarily designed to help projects to adapt to change requests.
- Agile approaches are based on some common principles, some of which are;
  - Working software is the key measure of progress in a project.
  - Software should be developed and delivered rapidly in small increments.
  - Even late changes in the requirements should be entertained.
  - Face-to-face communication is preferred over documentation.
  - Continuous feedback and involvement of customer is necessary for developing good-quality software.
  - Simple design which evolves and improves with time is a better approach than doing an elaborate design for handling all possible scenarios.
  - The delivery dates are decided by empowered teams of talented individuals.
- Some popular agile methodologies are XP (Extreme programming), Scrum, Unified Process, Crystal, DSDM and Lean.

Extreme programming (XP)

- An extreme programming project starts with user stories.
- User stories are the short descriptions of the requirements.
- They are different from traditional requirements specification where each requirement is specified in greater detail.
- For example, the following figure shows user stories.



- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.

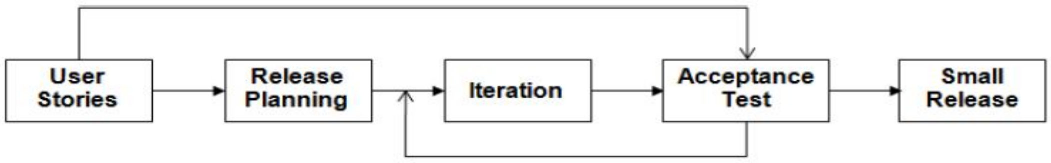
▪ The development team estimates how long it will take to implement a user story. Generally, implementation of a user story takes one to four weeks.

▪ Using these estimates and the stories, release planning is done which defines which stories are to be built in which system release, and the dates of these releases.

▪ Usually, small releases are encouraged and each release is done in some iterations.

▪ Acceptance tests are performed to test the software before the release.

▪ The overall process of extreme programming is shown below.



▪ As shown in the above figure, development is done in iterations by considering the user stories one by one.

▪ An iteration starts with iteration planning in which the stories to be implemented in this iteration are selected.

▪ At a time, only one user story is planned, developed and tested using acceptance testing.

▪ An iteration may not add significant functionality but, still a new release is made at the end of each iteration.

▪ After some iterations, the entire software is released.

Advantages

- Customer satisfaction by rapid, continuous delivery of useful software.
- Customers, developers and testers constantly interact with each other.
- Working software is delivered frequently (weeks rather than months).
- Facilitates face-to-face conversation.
- Provides close and daily cooperation between business people and developers.
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed.

Disadvantages

- Can be misinterpreted…
- Difficult to get external review.
- When project is complete, and team disperses, maintenance becomes difficult.

Where to use Agile Process Model

- XP, and other agile methods, are suitable for situations where the volume and pace of requirements change is high, and where requirement risks are considerable.
- It works well when the customer is willing to involve heavily during the entire development, working as a team member.

UNIT-II
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) Which of the following is a characteristic of Agile development? [ ]

a) Shared code ownership

b) Implement the simplest solution to meet today's problem

c) Continual feedback from customer

d) test-driven development

e) All of the above

2) In waterfall model, output of one phase is input to next phase. [ ]

a) True b) False

3) If requirements are easily understandable and defined then which model is best suited? [ ]

a) Waterfall Model b) Iterative Development Model

c) Prototyping d) Extreme Programming

4) Which of the following are advantages of iterative model? [ ]

a) Early revenue generation

b) Simpler to manage

c) Divided workload

d) Early feedback

e) All the above

5) Which phase of the RUP is used to establish a business case for the system [ ]

a) Transition b) Elaboration

c) Construction d) Inception

6) In XP Increments are delivered to customers every _______ weeks. [ ]

a) One b) Two c) Three d) Four

7) In a college, students are asked to develop a software. Which model would be Preferable? [ ]

a) Waterfall model b) Spiral model

c) Agile model d) Code and fix model

8) Which of the following life cycle model can be chosen if the development team has less experience on similar projects? [ ]

a) Spiral b) Waterfall

c) Prototyping d) Iterative Enhancement Model

9) Which four framework activities are found in the Extreme Programming(XP)?

a) analysis, design, coding, testing [ ]

b) planning, analysis, design, coding

c) planning, design, coding, testing

d) planning, analysis, coding, testing

10) An iterative process of system development in which requirements are converted to a working system that is continually revised through close work between an analyst and user is called__________________ [ ]

a) Waterfall modeling

b) Iterative modeling

c) Spiral modeling

d) None of these above

11) A company is developing an advance version of their current software available in the market, what model approach would they prefer ? [ ]

a) waterfall

b) Prototyping

c) Iterative Enhancement

d) Spiral

12) Which one of the following statements most accurately identifies the stakeholders in a software development project? [ ]

a) A stakeholder of the organization developing the software

b) Anyone who is interested in the software

c) Anyone who is a source of requirements for the software

d) Anyone who might be affected by the software

SECTION-B

SUBJECTIVE QUESTIONS

1) What is a process model? How do process models differ from one another?
2) What is the oldest paradigm for software engineering? Why does the waterfall model sometimes fail?
3) Write about the Rational unified process model in detail.
4) Compare the waterfall model with the Unified process model.
5) Explain about agile methodology & extreme programming as software development process models.
6) Describe prototyping model in detail. Discuss how to select a particular process model based on characteristics of a project.
7) Categorize the strengths and weaknesses of waterfall, iterative development and prototyping.
8) Analyze why does iterative process makes it easier to manage change.
9) Is it possible to combine the process models? If so explain with an example.
10) Which process model is suitable for medium scale projects, justify it.

SECTION-C

GATE QUESTIONS

1) Match the following: [ ]

1. Waterfall Model	a) Specifications can be developed incrementally
2. Evolutionary Model	b) Requirements compromises are inevitable
3. Component-based Software Engineering	c) Explicit recognition of risk
4. Spiral Development	d) Inflexible partitioning of the project into stages

(a) 1-a, 2-b, 3-c, 4-d **(b)** 1-d, 2-a, 3-b, 4-c
(c) 1-d, 2-b, 3-a, 4-c **(d)** 1-c, 2-a, 3-b, 4-d **(Gate CS 2015)**

2) Which one of the following is TRUE? [ ]

(a) The requirements document also describes how the requirements that are listed in the document are implemented efficiently.

(b) Consistency and completeness of functional requirements are always achieved in practice.

(c) Prototyping is a method of requirements validation.

(d) Requirements review is carried out to find the errors in system design **(GATE CS 2014)**

3) What is the appropriate pairing of items in the two columns listing various activities encountered in a software life cycle? [ ]

P. Requirements Capture	1. Module Development and Integration
Q. Design	2. Domain Analysis
R. Implementation	3. Structural and Behavioral Modeling
S. Maintenance	4. Performance Tuning

a) P-3, Q-2, R-4, S-1 **(b)** P-2, Q-3, R-1, S-4

(c) P-3, Q-2, R-1, S-4 **(d)** P-2, Q-3, R-4, S-1 **(GATE CS 2010)**

UNIT –III

Planning a Software Project

3.1 Effort estimation

- Effort and schedule estimates are essential prerequisites for planning the project.
- Without these, even simple questions like
  - ✓ is the project late?
  - ✓ are there cost overruns? And
  - ✓ when is the project likely to complete?

  Cannot be answered.
- For software development human resources are needed. Therefore most cost estimation procedures focuses on estimating effort in terms of person-months.
- Software requirement analysis plays an important role in effort estimation.
- There are 2 popular methods for effort estimation
  1. Top down estimation approach
  2. Bottom up estimation approach

3.1.1 Top down estimation approach

- The top down estimation approach considers effort as the function of Size. The larger the project, the greater is the effort requirement.
- Let P=productivity (in KLOC/PM) then the effort estimate for the project will be SIZE/P person months (PM).
- But this approach can work only if the size and type of the project are similar.
- A more general function for determining effort from size that is commonly used is of the form

$$\textbf{Effort} = a \times \textbf{SIZE}^{b}$$

  - ✓ where a and b are constants, and
  - ✓ project size is generally in KLOC

- Values for these constants for an organization are determined through regression analysis.
- The top-down approach is normally associated with parametric models such as function points and COCOMO.
- Having calculated the overall effort required, the problem is then to allocate proportions of that effort to the various activities within that project.

Advantages

- Very useful when cost estimates are needed in the very early phases of a project.
- It can be used to estimate the effort even when you have less information available on the project.
- It allows refining your earlier estimates and adding detail as the project moves forward.

Disadvantages

- Less accurate than other estimating techniques.
- Provides less scope to get lower levels of input.

3.1.2 Bottom up estimation approach

- In this approach, the project is first divided into tasks and then estimates for the different tasks of the project are obtained.
- From the estimates of the different tasks, the overall estimate is determined.
- This type of approach is also called as **Activity-Based Estimation**.
- Once the project is partitioned into smaller tasks, it is possible to directly estimate the effort required for them, especially if tasks are relatively small.
- The procedure for estimation can be summarized as follows:
  1. Identify modules in the system and classify them as simple, medium, or complex.
  2. Determine the average coding effort for simple/medium/complex modules.

3. Get the total coding efforts using the coding effort of different types of modules and the counts for them.
4. Using the effort distribution for similar projects, estimate the effort for other tasks and the total effort.
5. Refine the estimates based on project-specific factors.

Advantages

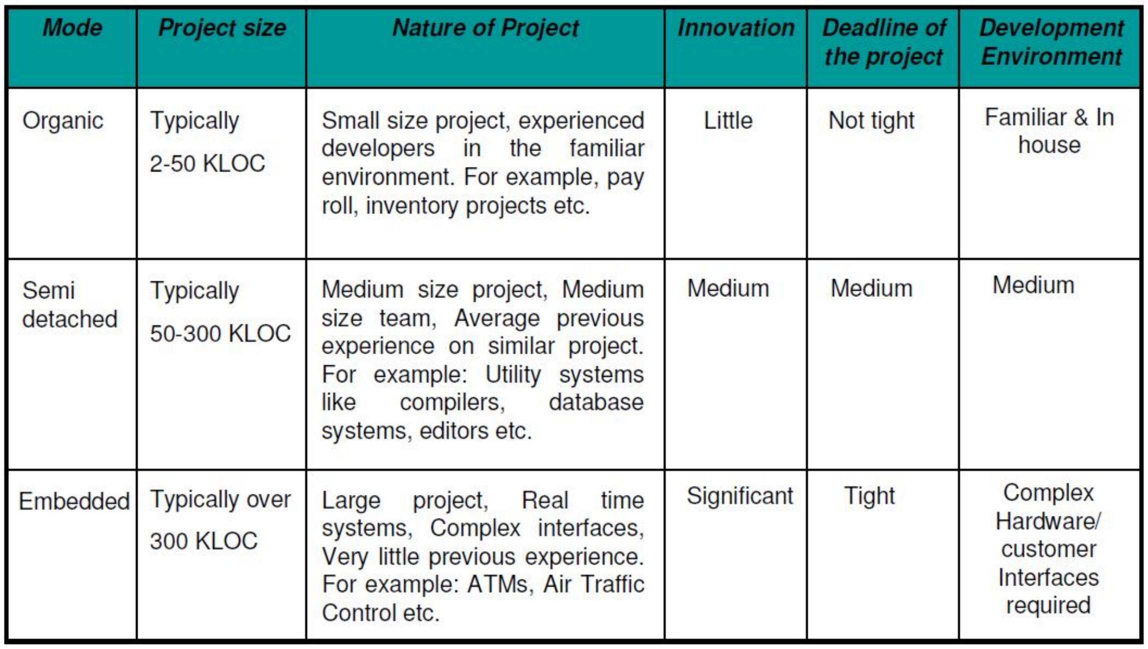
- Greater accuracy.
- It is simple and easy.

Disadvantages

- Directly estimating the overall effort is not possible, because some tasks may omit some activities.

3.1.3 COCOMO Model

- COCOMO (Constructive COst Model) is an algorithmic cost estimation technique proposed by Boehm.
- It is designed to provide some mathematical equations to estimate software projects.
- These mathematical equations are based on historical data and use project size in the form of KLOC.
- The COCOMO model is a multi-variable model and uses several variables for effort estimation.
- The value of these parameters depends on the project mode (or type) shown in the following table.



Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

- There are three variations of COCOMO model.
    1. Basic COCOMO
    2. Intermediate COCOMO and
    3. Detailed COCOMO

3.1.3.1 Basic COCOMO Model

- The Basic COCOMO Model estimates effort as a function of size measured in KLOC.
- The basic COCOMO Model is very simple, quick, and applicable to small to medium organic-type projects.
- It is given as follows:

$$\textbf{Development Effort (E) = a x (KLOC)}^{b}\textbf{ PM}$$

$$\textbf{Development Time (T) = c x (E)}^{d}\textbf{ Months}$$

- Where a, b, c, and d are constants and these values are determined from the historical data of the past projects.
- The values of a, b, c and d for different types of projects are shown in table below.

Project Category	a	b	c	d
Organic	3.2	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Example 3.1

Assume that a system for student course registration is planned to be developed and its estimated size is approximately 10,000 lines of code. The organization is proposed to pay Rs. 25000 per month to software engineers. Compute the development effort, development time, and the total cost for product development.

Solution

The project can be considered an organic project since, its size (10 KLOC) lies between 2-50 KLOC. Thus, from the basic COCOMOmodel,

Development Effort (E) = $3.2 \times (10)^{1.05} = 35.90$ PM

Development Time (T) = $2.5 \times (35.90)^{0.38} = 9.747$ Months

Total Product Development Cost

= Development Time *Salaries of Engineers

= 9.747×25000

= Rs. 2,43,675/-

3.1.3.2 Intermediate COCOMO Model

- The basic COCOMO model determines the effort and time based on the project size.
- There are certain other factors that can affect the project size and hence the development effort and time.
- Therefore, Boehm has introduced 15 cost drivers, considering the various aspects of product development environment.
- These cost drivers are used to adjust the project complexity and are termed as effort adjustment factors (EAF).
- These cost drivers are classified as Computer Attributes, Product Attributes, Project Attributes, and Personnel Attributes.

- The intermediate COCOMO model computes software development effort as a function of the program size and a set of cost drivers. The cost drivers and their ratings are shown in Table 3.1.
- The intermediate COCOMO model estimates the initial effort using the basic COCOMO model.
- Then the EAF is calculated as the product of 15 cost drivers.
- Total effort is determined by multiplying the initial effort with the total value of EAF.
- The computation steps are summarized below.

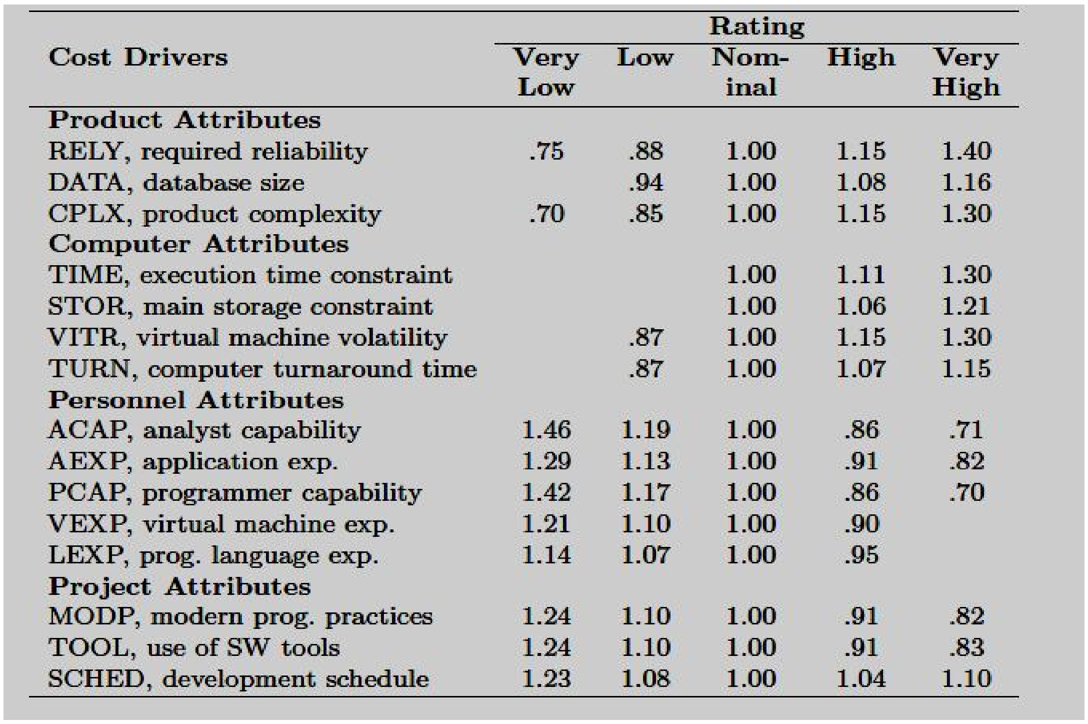
Development effort (E):
Initial effort (E_i) = $\mathbf{a \times (KLOC)^{b}}$

Effort Adjustment Factor (EAF) = $EAF_1 \times EAF_2 \times \ldots \times EAF_n$

Total development effort (E) = $E_i \times EAF$

Development time (T) = $\mathbf{c \times (E)^{d}}$

Table 3.1: Cost drivers and their values



Cost Drivers	Rating				
	Very Low	Low	Nom-inal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

Example 3.2

Suppose a library management system (LMS) is to be designed for an academic institution. From the project proposal, the following five major components are identified:

Online data entry	-	1.0 KLOC
Data update	-	2.0 KLOC
File input and output	-	1.5 KLOC
Library reports	-	2.0 KLOC
Query and search	-	0.5 KLOC

The database size and application experience are very important in this project. The use of the software tool and the main storage is highly considerable. The virtual machine experience and its volatility can be kept low. All other cost drivers have nominal requirements. Use the COCOMO model to estimate the development effort and the development time.

Solution

The LMS project can be considered an organic category project. The total size of the modules is 7 KLOC. The development effort and development time can be calculated as follows:

Development effort

Initial effort (E_i) $= 3.2\times(7)^{1.05}$

$= 24.6889$ **PM**

Effort Adjustment Factor (EAF)

$= 1.16 \times 0.82 \times 0.91 \times 1.06 \times 1.10 \times 0.87 = 0.8780$

Total effort (E) $= E_i * EAF$

$= 24.6889 \times 0.8780$

$= 21.6785$ **PM**

Development time (T) $= 2.5 \times (E)^{0.38}$ **Months**

$= 2.5\ (21.6785)^{0.38}$ **months**

$= 8.0469$ **months**

3.1.3.3 Detailed COCOMO Model

- The detailed COCOMO inherits all the features of the intermediate COCOMO model for the overall estimation of the project cost.
- For detailed planning and scheduling, it is necessary to assess the impact of the cost drivers in a phased manner.
- The detailed COCOMO model uses different cost drivers for each phase of the project. Phase-wise effort multipliers provide better estimates than the intermediate model.
- The detailed COCOMO model defines five life cycle phases for effort distribution:
    - Plan and Requirement,
    - System Design,
    - Detailed Design,
    - Code and Unit Test, and
    - Integration Test.
- The percentage of effort in a phase varies with the project size and the project nature. The phase-wise percentage of effort distribution for some KLOC values is shown in table 3.2.
- In the detailed COCOMO model, effort is calculated as a function of size (in terms of KLOC) and the value of a set of cost drivers according to each phase of the software life cycle.

Example

Compute the phase-wise development effort for the problem discussed in Example 3.2 above.

Solution

There are five components in the organic project discussed in Example above: Online Data Entry, Data Update, File Input and Output, Library Reports, Query and Search.

The estimated effort (E) is 21.6785 PM. The total size is 7 KLOC, which is between 2 KLOC and 32 KLOC. Thus, the actual percentage of effort can be calculated as follows:

H.KLOC DV + [(L.KLOC DV - H.KLOC DV) / (H.KLOC - L.KLOC)]*Size

Plan and Requirement (%) = 6 + (6 - 6) / (32 - 2) × 7 = 6%

Effort = 0.06 × 21.6785 PM = 1.30071 PM

System Design = 16 + (16 - 16) / (32 - 2) × 7 = 16%

Effort = 0.16 × 21.6785 PM = 3.46856 PM

Detailed Design = 24 + (26 - 24) / (32 - 2) × 7 = 25%

Effort = 0.25 × 21.6785 PM = 5.419625 PM

Code and Unit Test = 38 + (42 - 38) / (32 - 2) × 7= 39%

Effort = 0.39 × 21.6785 PM = 8.454615 PM

Integration Test = 22 + (16 - 22) / (32 - 2) × 7= 24%

Effort = 0.24 × 21.6785 PM =5.20284 PM

Table 3.2: Phase-wise Effort Distribution for Detailed COCOMO Model

Project type and size	Plan and requirement	System design	Detailed design	Code and unit test	Integration and test
Percentage-wise distribution of development effort					
Organic (2 KLOC)	6	16	26	42	16
Organic (32 KLOC)	6	16	24	38	22
Semidetached (32 KLOC)	7	17	25	33	25
Semidetached (128 KLOC)	7	17	24	31	28
Embedded (128 KLOC)	8	18	25	26	31
Embedded (320 KLOC)	8	18	24	24	34
Percentage-wise distribution of development time					
Organic (2 KLOC)	10	19	24	39	18
Organic (32 KLOC)	12	19	21	34	26

Semidetached (32 KLOC)	20	26	21	27	26
Semidetached (128 KLOC)	22	27	19	25	29
Embedded (128 KLOC)	36	36	18	18	28
Embedded (320 KLOC)	40	38	16	16	30

3.4 Project Schedule and Staffing

- Project scheduling is one of the important activities done by the project manager in order to complete the project successfully.
- Project scheduling involves planning of various activities that need to be accomplished when, how and by whom. These activities are as follows.
  - Task identification
  - Work breakdown structure
  - Task interdependency identification
  - Time allocation
  - Resource allocation
  - Monitoring, tracking and control
- For a project with some estimated effort, multiple schedules (or project duration) are possible.
- For example, if a project whose effort estimate is 56 person-months, then
  - ✓ A schedule of 8 months is possible with 7 people.
  - ✓ A schedule of 7 months with 8 people is also possible,
  - ✓ A schedule of approximately 9 months with 6 people is also possible.
  - ✓ But a schedule of 1 month with 56 people is not possible.
  - ✓ Similarly, no one would execute the project in 28 months with 2 people.
- Hence, the objective is to fix a reasonable schedule to do the project.

- One method to determine the overall schedule is to determine it as a function of effort.
- The IBM Federal Systems Division found that the total duration, M, in calendar months can be estimated by **$M = 4.1*E^{0.36}$** .
- In COCOMO, the equation for schedule for an organic type of software is **$M = 2.5*E^{0.38}$** .
- Another method for determining a schedule for medium-sized projects is the rule of thumb called the **Square Root Check**. In this, the schedule can be determined by computing the square root of the total effort in person months.
- Once the schedule is determined, then it is required to determine the number of people required in each phase of the project development.
- Usually, number of people can be utilized in a software project tends to follow the Rayleigh curve as shown below.
- It says that, in the beginning and the end, few people are needed on the project; the peak team size (PTS) is needed somewhere near the middle of the project; and again fewer people are needed after that.

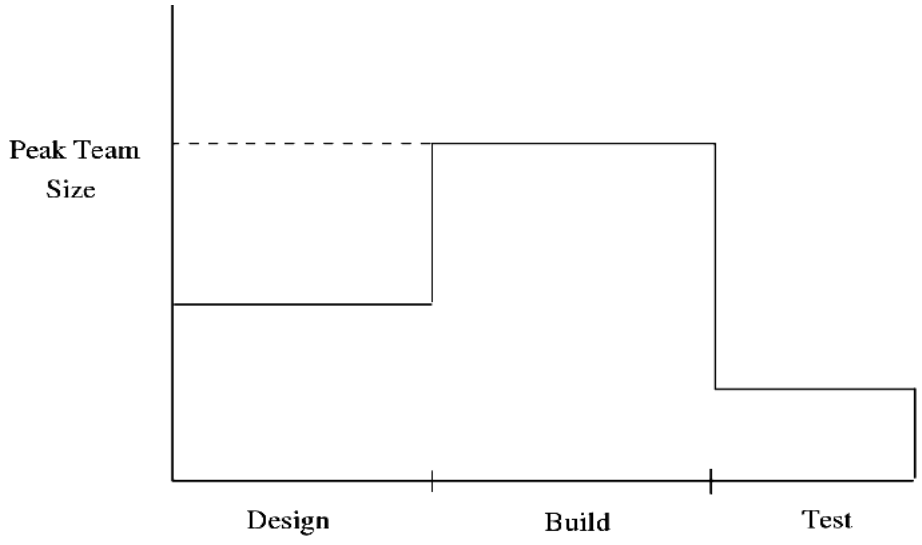


Figure 3.2: **Manpower ramp-up in a typical project**

- This occurs because only a few people can be used in the initial phases of requirements analysis and design. The human resources requirement peaks during coding and unit testing, and during system testing and integration, again fewer people are required.

- Generally speaking, design requires about a quarter of the schedule, build consumes about half, and integration and system testing consume the remaining quarter. COCOMO gives 19% for design, 62% for programming, and 18% for integration.

3.5 Quality Planning

- Software productivity and quality are the most considerable challenges in software development.
- It is easy to set goals for schedule and effort, but setting a goal for quality and then planning to meet it is very much difficult.
- Hence, often, quality goals are specified in terms of acceptance criteria.
- One acceptance criterion is the number of defects found during the acceptance testing.
    - Eg. The no.of defects found in software during acceptance testing should not exceed 100.
- Software development is a highly people-oriented activity and hence it is error-prone.
- The quality of the software product is determined based on the number of defects it has.
- Therefore, to plan for quality, let us first understand the defect injection and removal cycle because defects can be identified by injecting the known bugs into software. The defect injection and removal cycle is shown in figure 3.3.
- In a software project, we start with no defects. Then defects are injected into the software being built during different phases in the project. i.e. defects are injected in the requirements specification, the high-level design, the detailed design, and coding.

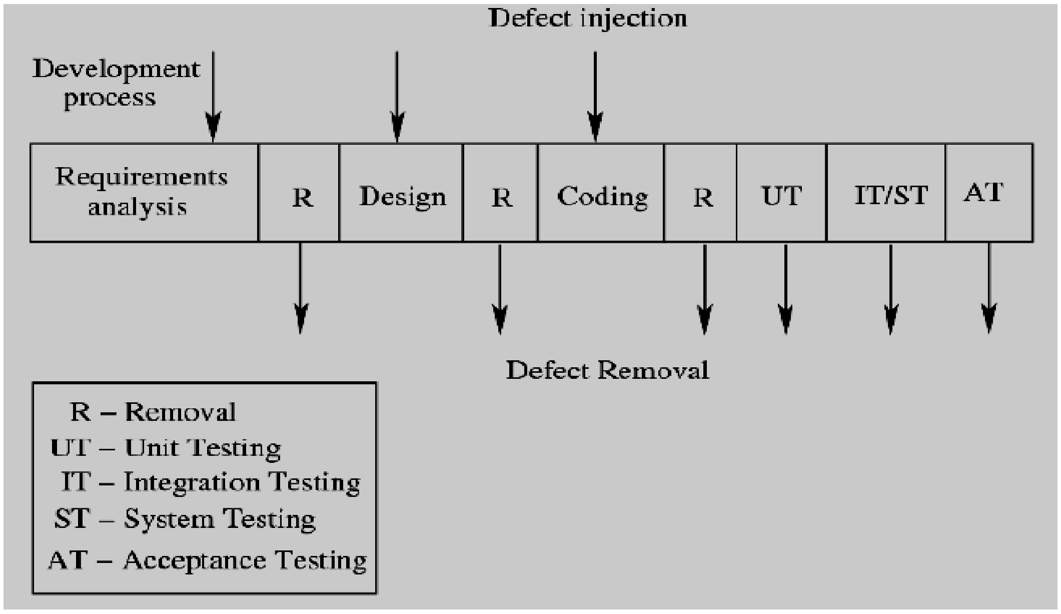


Figure 3.3: **Defect injection and removal cycle.**

- The above figure shows that defects are injected during requirements analysis, design and coding and then defects are identified and removed using QC activities.
- The QC activities for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, acceptance testing, etc.
- Hence, ensuring quality revolves around two main themes:
  - Reduce the defects being injected, and
  - Increase the defects being removed.
- The first is often done through standards, methodologies, following of good processes, etc., which help reduce the chances of errors by the project personnel.
- The quality plan therefore focuses mostly on planning suitable quality control tasks for removing defects.
- Reviews and testing are two most common QC activities utilized in a project to remove the defects.

- Hence, the quality plan for the project is a specification of which QC task is to be done and when, and what process and guidelines are to be used for performing the QC task.
- Typically, the QC tasks will be schedulable tasks in the detailed schedule of the project. For example, it will specify what documents will be inspected, what parts of the code will be inspected, and what levels of testing will be performed.

3.6 Risk Management

- Software projects and businesses are full of uncertainties due to changing requirements, varying range of applications, informal processes and technological advancements.
- Risk is an unfavourable situation that may lead to undesirable outcome.
- Risk management is a proactive approach for minimizing the uncertainty and potential problems associated with a project.
- Software risk management is necessary
  - To reduce the rework caused by missing, erroneous, or ambiguous requirements, design or code.
  - To avoid software project disasters including overrun of budgets and schedules, defect-driven software and operational failures.
  - To stimulate a win-win software solution where customers receive the product they need and the vendors make the profits they except.
  - To keep provision for the detection and prevention of risks.
- There are typically 3 types of risk occurrences.
  1. **Project risks** – are concerned with project planning and scheduling. Most of the projects run behind schedule, become over budget and have lack of skilled engineers.
  2. **Product risks** – are concerned with design and development of software products. These risks affect product quality and its

performance. These risks include technology obsolescence, changing requirements, technical uncertainty, excessive constraints, lack of technical knowledge etc.

3. **Business risks** – are the negative impacts on the operation or profitability of an organization. Eg. An organization is unable to produce products which are demanded in the market; the competitor has launched an alternative product etc. Business risks are mainly due to wrong decisions.

3.6.1 Risk Management Activities

- In order to handle all the above risks, some systematic process of risk management is required. The risk management process includes the following risk management activities.

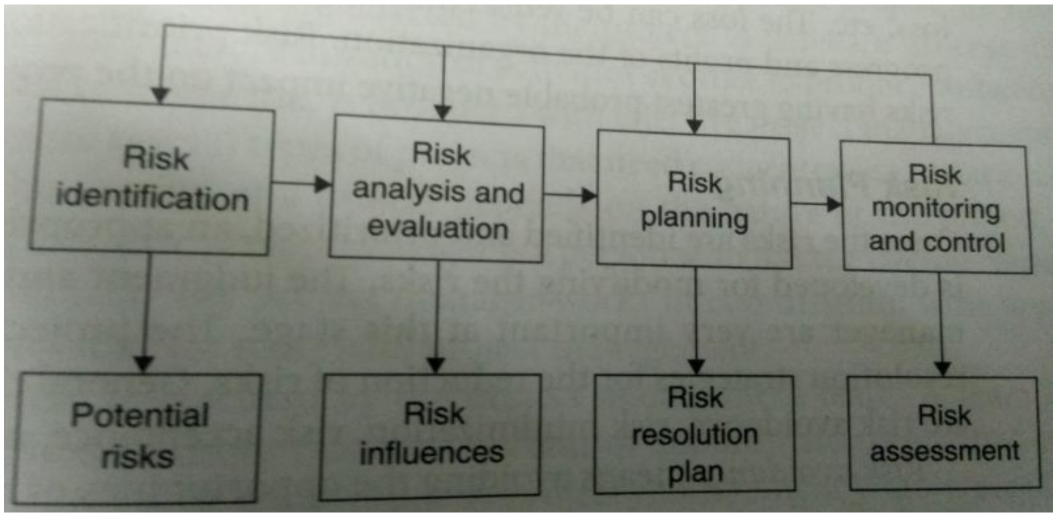


Figure 3.4: **Risk Management Activities**

Risk Identification

- In this potential risks that affect the software product are identified along with their sources, root causes, consequences and types.
- Different types of risks may arise. Eg. Requirements risks, technology risks, organizational risks, tool risks, human resources risks, estimation risks etc.

- Different techniques are used to identify the risks, such as interviewing, brainstorming, assumption analysis, critical path analysis, utilization of risk taxonomies etc.
- Risk identification is done based on the historical data.

Risk Analysis

- In this each risk is analyzed independently by examining and assessing its impact, probability, and seriousness.
- It can be done using different techniques like metrics, decision trees and scenario analysis.
- The list of risks is then grouped and prioritized/ranked based on the risk analysis. It helps in resource allocation and management.
- Boehm defined the risk exposure (RE) to establish risk priorities. It measures the impact of a risk in terms of loss.
- RE is defined as the probability of an undesired outcome times the expected loss if that outcome occurs.

  RE = Prob(UO) * Loss(UO)

  Where Prob(UO) is the probability of occurrence of the undesirable outcome and

  Loss(UO) is the total loss incurred due to that undesirable outcome.

Risk Planning

- Once the risks are identified and prioritized, an appropriate management policy is developed to handle the risks.
- The project manager uses risk resolution strategies for reducing the risks. These are
- **Risk avoidance** – avoiding the risks. It is the most effective way. Project managers should not take decisions of working in a new way that may increase the risk.
  - Eg. Not paying money by debit card.
- **Risk minimization** – focus on identifying the options for reducing the likelihood or consequences of the risk.

  - Eg. Backup programmers must be recruited to continue the project in case of people working on the project leave the organization.
- **Risk acceptance** – comes when you have no option other than accepting the risk. In this case, project manager takes executive support to mitigate the risk.
- **Risk transfer** – transfer the risk to somebody else. This can be done by choosing outsourcing, subcontracting, buying a resource or tool.

Risk Monitoring and Control

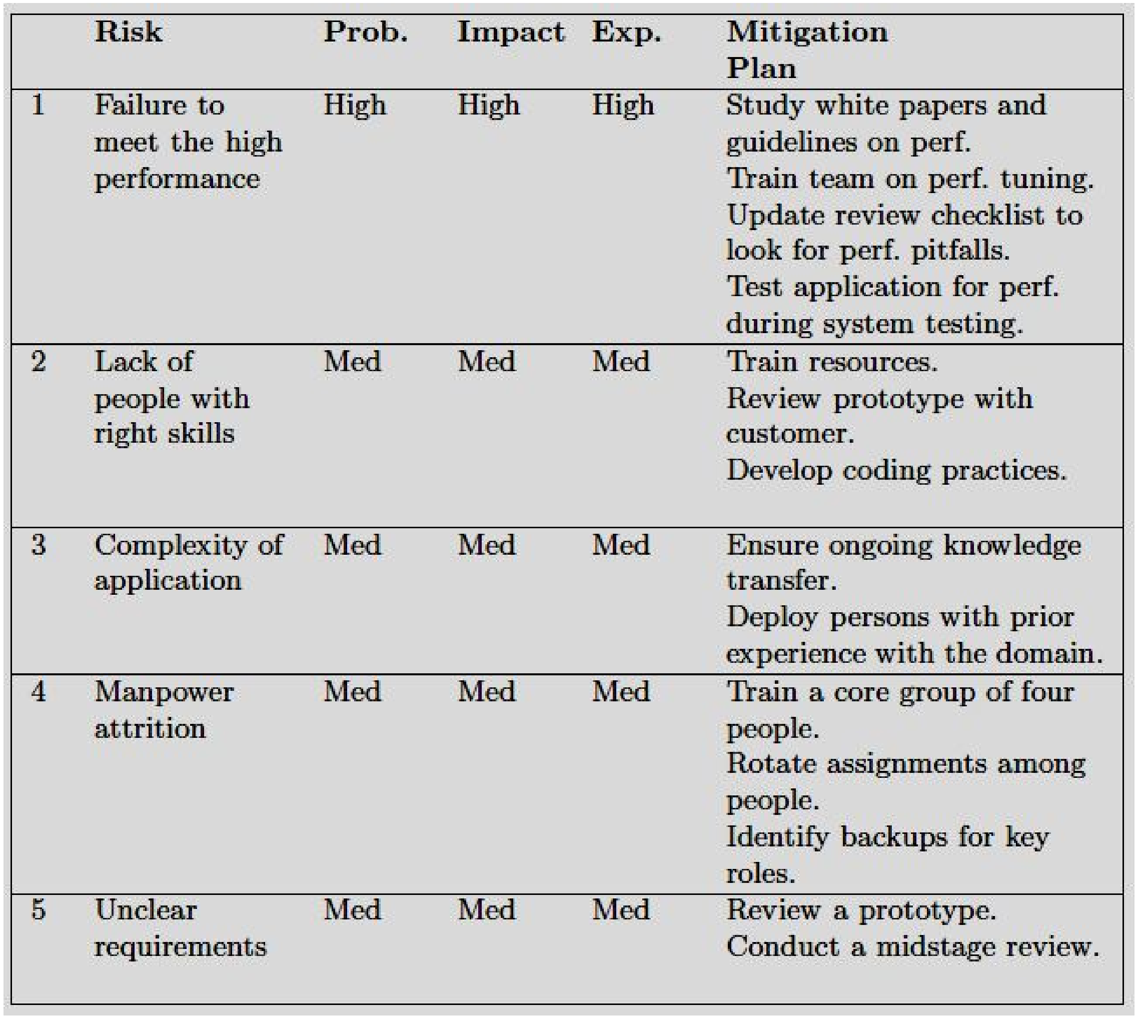
- Risk monitoring and control ensures new risks are detected and managed.
- Risk action plans are implemented to reduce the impact of risks.
- Policies and standards are regularly carried out to identify the opportunities for improvement.
- If needed, changes are made in the organizational environment to cope with risks.

3.6.2 A Practical Risk Management Planning Approach

- In this approach, the probability of a risk occurring is categorized as low, medium, or high. The risk impact can also be classified as low, medium, and high. With these ratings, the following simple method for risk prioritization can be specified:

1. For each risk, rate the probability of its happening as low, medium, or high.
2. For each risk, assess its impact on the project as low, medium, or high.
3. Rank the risks based on the probability and effects on the project; for example, a high-probability, high-impact item will have higher rank than a risk item with a medium probability and high impact. In case of conflict, use judgment.
4. Select the top few risk items for mitigation and tracking.

- The following figure shows a practical risk management plan for a typical project.



	Risk	Prob.	Impact	Exp.	Mitigation Plan
1	Failure to meet the high performance	High	High	High	Study white papers and guidelines on perf. Train team on perf. tuning. Update review checklist to look for perf. pitfalls. Test application for perf. during system testing.
2	Lack of people with right skills	Med	Med	Med	Train resources. Review prototype with customer. Develop coding practices.
3	Complexity of application	Med	Med	Med	Ensure ongoing knowledge transfer. Deploy persons with prior experience with the domain.
4	Manpower attrition	Med	Med	Med	Train a core group of four people. Rotate assignments among people. Identify backups for key roles.
5	Unclear requirements	Med	Med	Med	Review a prototype. Conduct a midstage review.

Figure 3.5: Practical risk management plan for a project

Unit – IV

Software Requirements Analysis and Specification

4.1. Value of a Good SRS

- ✓ An SRS establishes the basis for agreement between the client and the supplier on what the software product will do.
  - i.e. it acts as the contract between the client (or the customer) and the developer (the supplier).
  - So, through SRS, the client clearly describes what he expects from the supplier, and the developer clearly understands what capabilities to build in the software.
- ✓ An SRS provides a reference for validation of the final product.
  - i.e. the SRS helps the client to determine whether the software meets the requirements.
  - Without SRS it is not possible for the client to check whether the software meets all the stated requirements in the SRS.
  - Similarly, without SRS it is not possible for the developer to convince the client that all the requirements have been fulfilled.
- ✓ A high-quality SRS is a prerequisite to high-quality software.
  - Many errors are made during the requirements phase. Good SRS can minimize changes and errors in the requirements.
  - Cost of fixing errors in requirements, design, coding, testing and maintenance increases exponentially.
- ✓ A high-quality SRS reduces the development cost.
  - Improving the quality of requirements substantially reduces the development cost.

4.2. Requirement Process

- The requirement process is the sequence of activities that need to be performed in the requirements phase in order to produce high quality SRS.
- The requirements process typically consists of three basic tasks:
  1. Problem or requirements analysis

2. Requirements specification and
3. Requirements validation

Problem or requirements analysis

- It often starts with a high-level problem statement.
- The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide.
- Frequently, during analysis, the analyst will have a series of meetings with the clients and end users.
- In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them.
- Any documents describing the work may be given along with the outputs of the existing methods of performing the tasks.
- In these early meetings, the analyst is basically the listener, absorbing the information provided.
- Once the analyst understands the system to some extent, he uses the next few meetings to seek clarifications of the parts he does not understand.
- In the final few meetings, the analyst essentially explains to the client what he understood about the system and cross checks the clients whether they are correct.

Requirements Specification

- After obtaining and analyzing the requirements, the next activity is to clearly specify the requirements.
- In this, we systematically organize the requirements.
- For many years requirements are specified using stories or scenarios or DFDs. But, today use case approach is followed to specify the requirements.
- Issues such as representation, specification languages, and tools are addressed during this activity.

Requirements validation

- It ensures that the SRS specifies all the requirements of the software and the SRS is of good quality.
- The requirements process ends with the production of a high quality SRS.

4.3. Requirement Specification

For many years requirements are specified using stories or scenarios. But, today use case approach is followed to specify the requirements. Before discussing the specification of requirements, let's see the following.

4.3.1 Desirable Characteristics of an SRS

To properly satisfy the basic goals, an SRS should have certain properties and should contain different types of requirements. Some of the desirable characteristics of an SRS are

1. **Correct** – The SRS is correct iff every requirement stated therein is one that the software shall meet. There is no tool or procedure that assures correctness.
2. **Complete** - The SRS is complete if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.
3. **Unambiguous** - the SRS is unambiguous if and only if every requirement stated has one and only one interpretation (or meaning).
4. **Verifiable** – the SRS is verifiable if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement.
5. **Consistent** – the SRS is consistent if there is no requirement that conflicts with another. For example, suppose a requirement states that an event e is to occur before another event f. But then another set of requirements states (directly or indirectly by transitivity) that event f should occur before event e. Such a requirement is said to be inconsistent requirement.

6. **Ranked for importance and/or stability** – the SRS is ranked for importance and/or stability if each requirement in it has an identifier which indicates the importance or stability of the requirement. Typically, all requirements are not equally important. In mission control applications, some requirements may be considered as more important and some requirements are considered as less important.

4.3.2 Components of an SRS

The basic issues an SRS must address are:

1. **Functionality** – specifies the functional requirements of the system. Functional requirement represents the service provided by the system. For example, the functional requirements of the library system are
   - Maintain books information
   - Maintain members information
   - Search book
   - Issue book
   - Return book etc.

Each functional requirement transforms a set of input data to corresponding output data.

For example, consider the functional requirement *Search Book* in a library system;

Functional Requirement: Search Book

Input: author's name

Output: details of books by that author and the locations of these books in the library

2. **Performance** – represents the non-functional requirements of the system.

   For library system, the non-functional requirements might be
   - Library system should respond to a query in less than 5 seconds
   - It shall operate with zero down time
   - It shall contain well written user manual

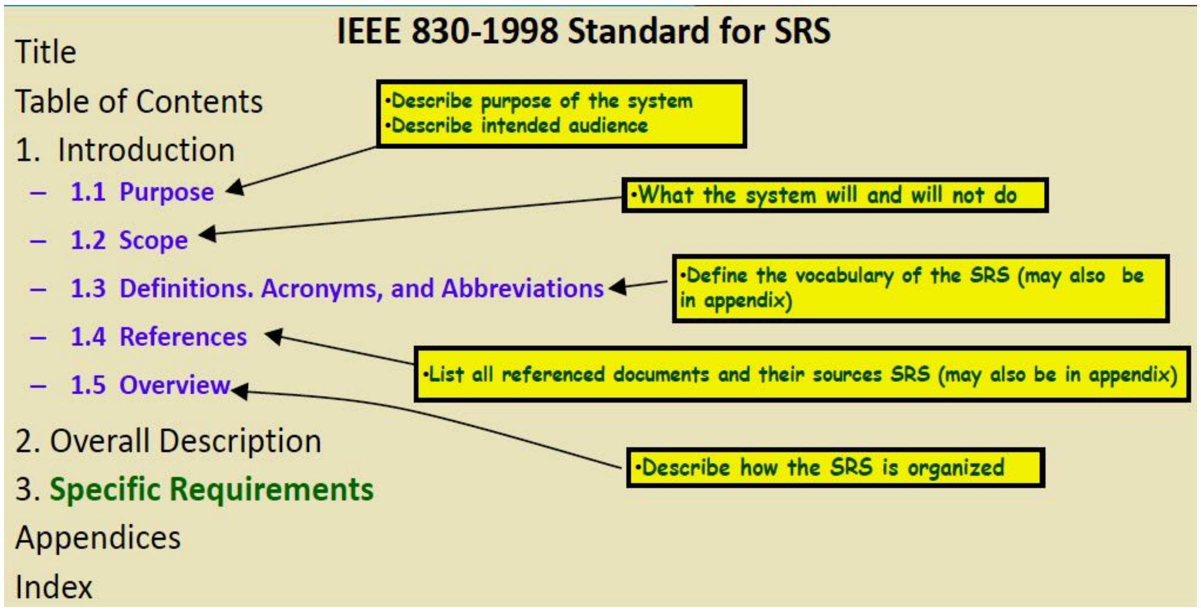
   - It shall allow upto 100 users to remotely connect to the system etc.

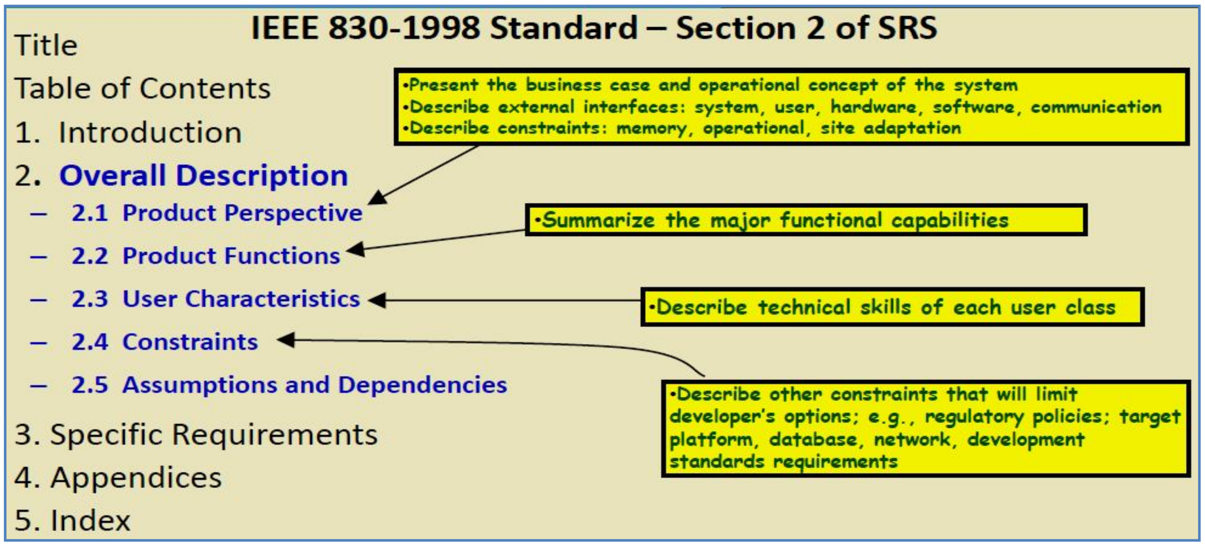
3. **Design constraints** – represents constraints imposed on the system For library system, the design constraints might be
   - It shall provide user interface through standard web browsers
   - It shall use an open source RDBMS such as Postgres SQL
   - It shall be developed using Java language etc.
4. **External interfaces** – It represents how does the software interacts with the people, the system's hardware, other hardware and other software. Examples of external interfaces are; GUI screens, file formats etc.

4.3.3 Structure of SRS Document

After collecting all data regarding the system to be developed,

- Remove all inconsistencies and anomalies from the requirements
- Systematically organize the requirements into a Software Requirements Specification (SRS) document.
- SRS document is known as black-box specification because it specifies what the system will do without specifying how it will do.
- Different templates are used by companies to specify SRS: Often variations of IEEE 830 standard is used for specifying SRS document.


IEEE 830-1998 Standard for SRS
Title
Table of Contents
1. Introduction
– 1.1 Purpose
– 1.2 Scope
– 1.3 Definitions. Acronyms, and Abbreviations
– 1.4 References
– 1.5 Overview
2. Overall Description
3. Specific Requirements
Appendices
Index
•Describe purpose of the system
•Describe intended audience
•What the system will and will not do
•Define the vocabulary of the SRS (may also be in appendix)
•List all referenced documents and their sources SRS (may also be in appendix)
•Describe how the SRS is organized



IEEE 830-1998 Standard – Section 2 of SRS

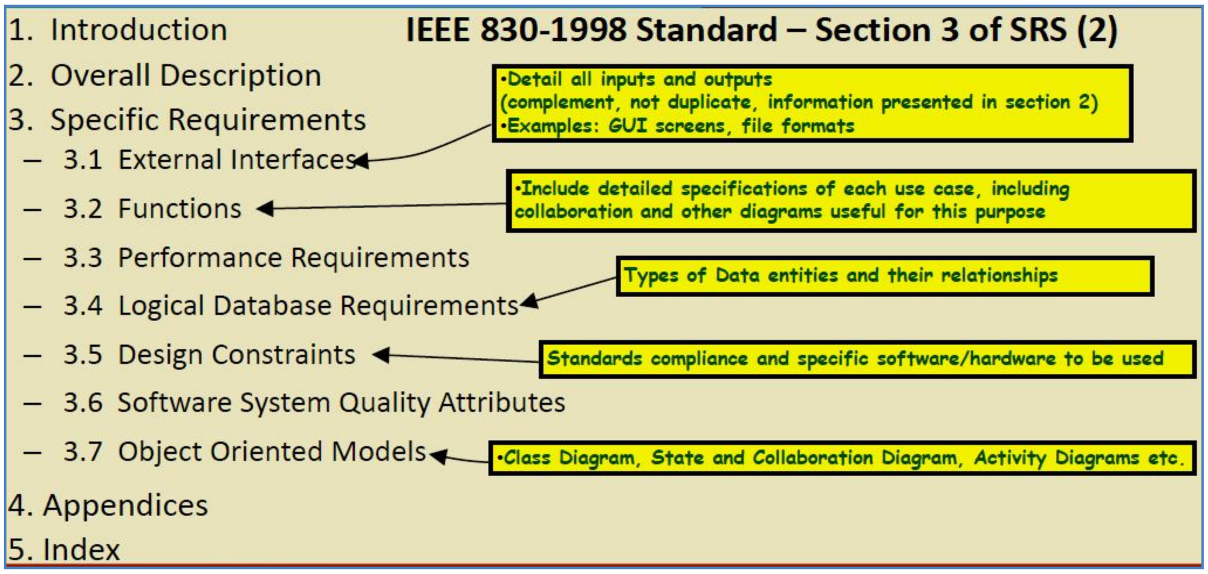
Title
Table of Contents
1. Introduction
2. Overall Description
– 2.1 Product Perspective
– 2.2 Product Functions
– 2.3 User Characteristics
– 2.4 Constraints
– 2.5 Assumptions and Dependencies
3. Specific Requirements
4. Appendices
5. Index

•Present the business case and operational concept of the system
•Describe external interfaces: system, user, hardware, software, communication
•Describe constraints: memory, operational, site adaptation

•Summarize the major functional capabilities

•Describe technical skills of each user class

•Describe other constraints that will limit developer's options; e.g., regulatory policies; target platform, database, network, development standards requirements



IEEE 830-1998 Standard – Section 3 of SRS (2)

1. Introduction
2. Overall Description
3. Specific Requirements
– 3.1 External Interfaces
– 3.2 Functions
– 3.3 Performance Requirements
– 3.4 Logical Database Requirements
– 3.5 Design Constraints
– 3.6 Software System Quality Attributes
– 3.7 Object Oriented Models
4. Appendices
5. Index

•Detail all inputs and outputs
(complement, not duplicate, information presented in section 2)
•Examples: GUI screens, file formats

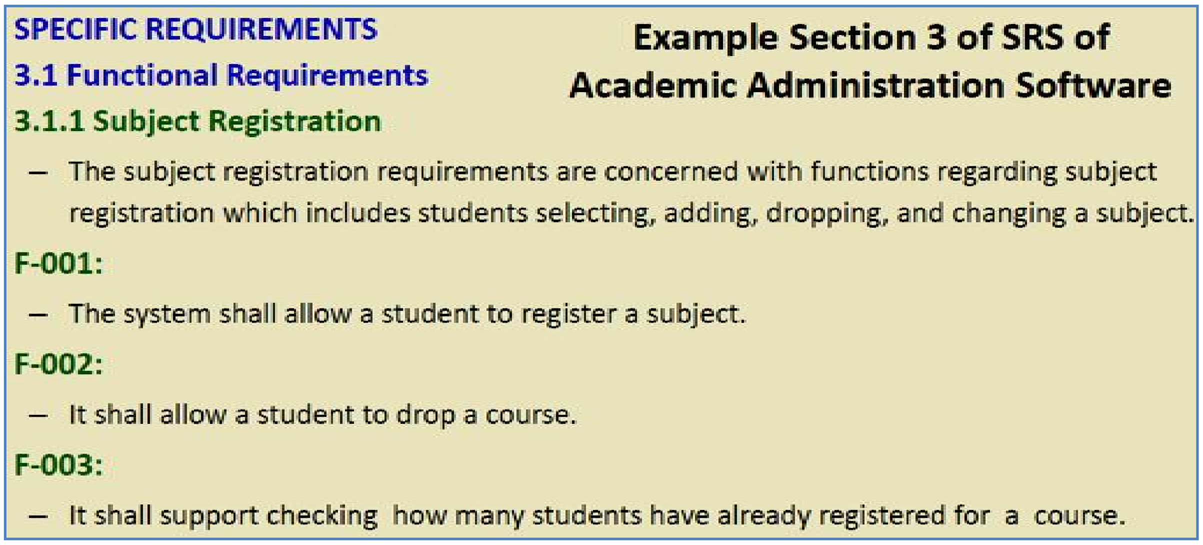
•Include detailed specifications of each use case, including collaboration and other diagrams useful for this purpose

Types of Data entities and their relationships

Standards compliance and specific software/hardware to be used

•Class Diagram, State and Collaboration Diagram, Activity Diagrams etc.

The following figure shows an example SRS document.



SPECIFIC REQUIREMENTS

Example Section 3 of SRS of Academic Administration Software

3.1 Functional Requirements

3.1.1 Subject Registration

– The subject registration requirements are concerned with functions regarding subject registration which includes students selecting, adding, dropping, and changing a subject.

F-001:

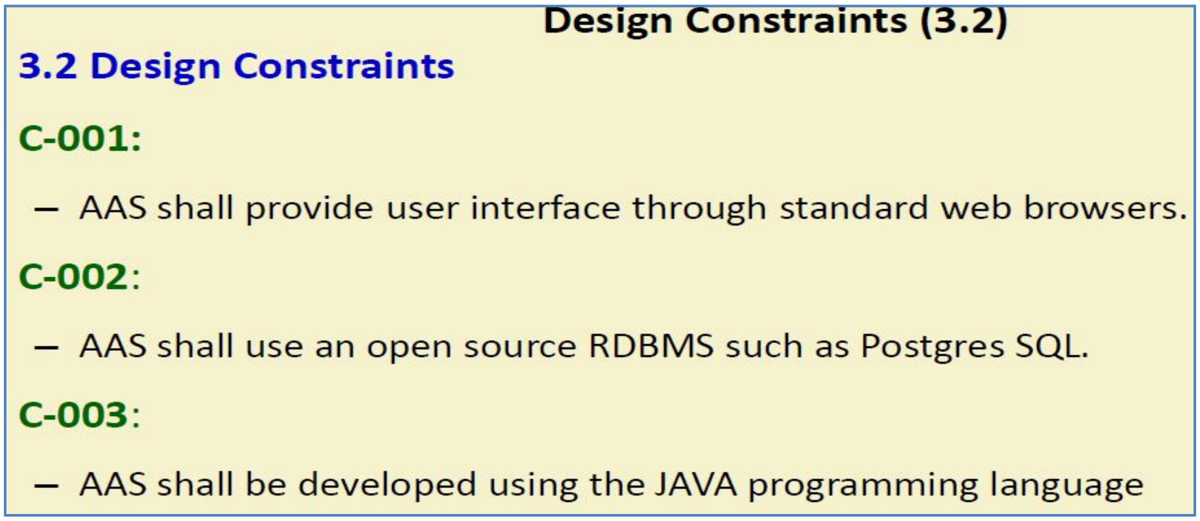
– The system shall allow a student to register a subject.

F-002:

– It shall allow a student to drop a course.

F-003:

– It shall support checking how many students have already registered for a course.



Design Constraints (3.2)

3.2 Design Constraints

C-001:

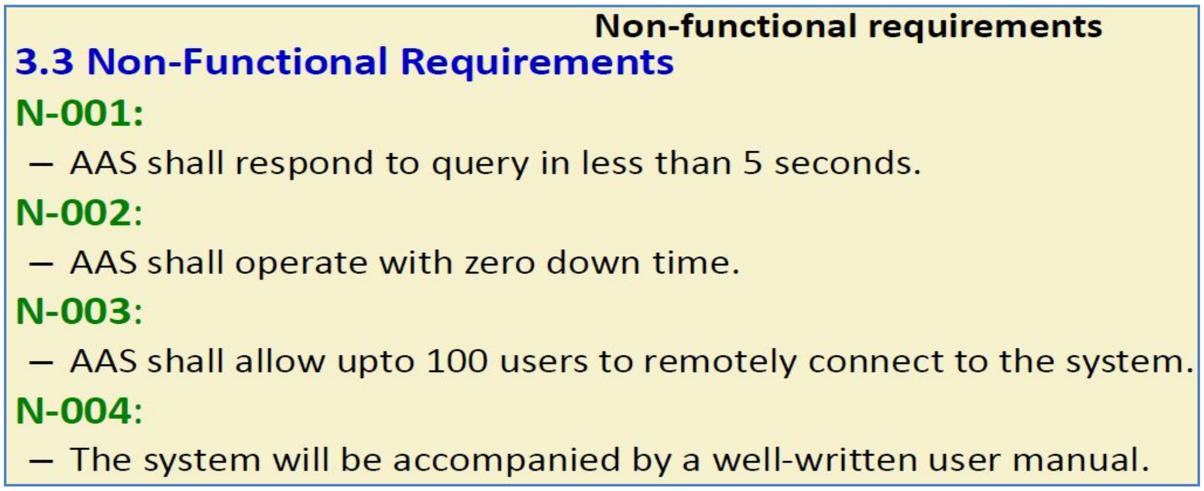
– AAS shall provide user interface through standard web browsers.

C-002:

– AAS shall use an open source RDBMS such as Postgres SQL.

C-003:

– AAS shall be developed using the JAVA programming language



Non-functional requirements

3.3 Non-Functional Requirements

N-001:

– AAS shall respond to query in less than 5 seconds.

N-002:

– AAS shall operate with zero down time.

N-003:

– AAS shall allow upto 100 users to remotely connect to the system.

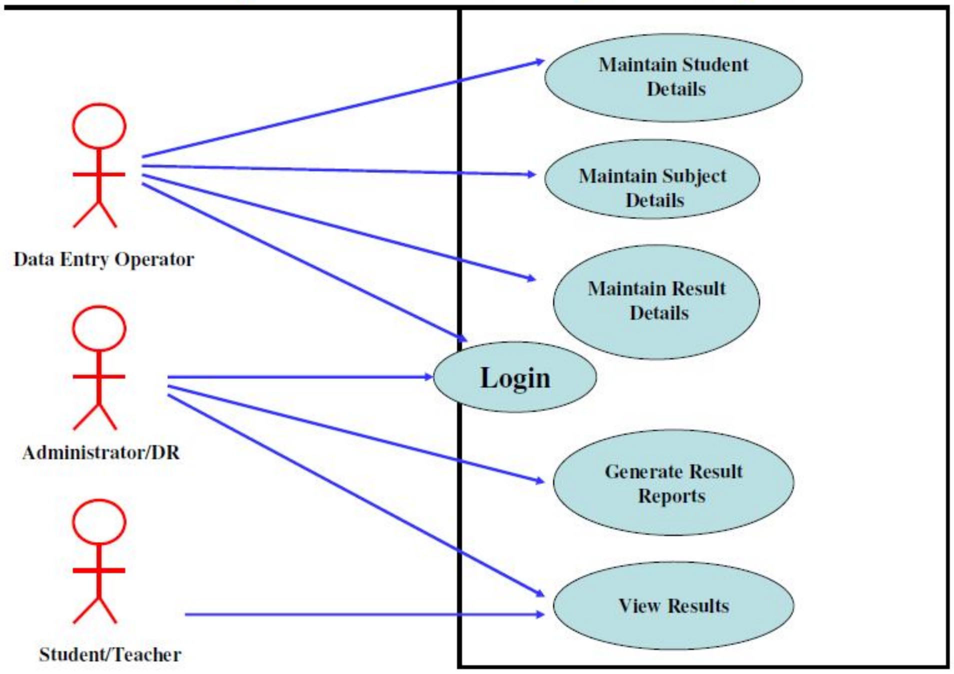
N-004:

– The system will be accompanied by a well-written user manual.

4.4 Functional Specification with Use Cases

- Ivar Jacobson & others introduced Use Case approach for gathering & modelling requirements.
- Use case represent high level functional requirement of a system.
- For example, the use cases of ATM system are;
  - Withdraw money
  - Deposit money
  - Transfer money are
  - Mini statement
  - Check balance etc.
- An actor initiates a use case for achieving a goal.
- An actor can be a person or a machine or another software system that interacts with this system.
- A uses case diagram is used to show the behaviour of a system. For example, the following figure shows the use case diagram for Result Management System.

Use case diagram for Result Management System



- Each use case is described by two types of scenarios.

- Main success scenario (or Basic flow)
- Exceptional scenarios (or Alternate flows)

▪ In addition to this, actor(s), preconditions and post conditions of the use case are also specified.

Example 4.1

Use Case – **Login**

1.1 **Actors:** 1. Data Entry Operator
2. Administrator/Deputy Registrar

1.2 **Pre conditions** – None

1.3 **Post conditions** - If the use case is successful, the actor is logged into the system. If not, the system state is unchanged.

1.4 **Basic Flow** - This use case starts when the actor wishes to login to the Result Management system.

i. System requests that the actor to enter user id and password.
ii. The actor enters user id & password.
iii. System validates user id & password, and if finds correct allow the actor to logs into the system.

1.5 **Alternate Flow**

1.5.1 If in the basic flow, the actor enters an invalid user id and/or password, the system displays an error message. The actor can choose to either return to the beginning of the basic flow or cancel the login, at that point, the use case ends.

Example 4.2

Use Case – **Withdraw Money (ATM System)**

1.1 **Actor** – Customer

1.2 **Pre conditions** –

1. ATM must be in a state ready to accept transactions
2. ATM must have at least some cash it can dispense
3. ATM must have at least some cash it can dispense

1.3 **Post conditions**

1. The current amount of cash in the user account is the amount before withdraw minus withdraw amount

2. A receipt was printed on the withdraw amount

1.4 **Basic Flow** – This use case starts when customer invokes withdraw money use case.

i. Customer inserts a Credit card into ATM
ii. System verifies the customer ID and status
iii. System asks for an operation type
iv. Customer chooses “Withdraw” operation
v. System asks for the withdraw amount
vi. Customer enters the cash amount
vii. System checks if withdraw amount is legal
viii. System dispenses the cash
ix. System deduces the withdraw amount from account
x. System prints a receipt
xi. Customer takes the cash and the receipt
xii. System ejects the cash card

1.5 **Alternate Flows**

1.5.1 Step 2: Customer authorization failed. Display an error message, cancel the transaction and eject the card.

1.5.2 Step 7: Customer has insufficient funds in its account. Display an error message, and go to step 5.

1.5.3 Step 7: Customer exceeds its legal amount. Display an error message, and go to step 5.

SE UNIT-IV
Assignment-Cum-Tutorial Questions

A) Objective Questions

1) What is the final outcome of requirements analysis and specification phase? [ ]

a) Drawing the data flow diagram b) The SRS document

c) Coding the project d) The user manual

2) Which of the following is not included in SRS document? [ ]

a) Functional requirements b)Non functional requirements

c) Goals of implementation d) User manual

3) As Software Manager, when you will decide the number of people required for a software project? [ ]

a) Before the scope is determined.

b) Before an estimate of the development effort is made

c) After an estimate of the development effort is made.

d) None of the above

4) Which of the following is not a 'concern' during the management of a software project? [ ]

a) Money d) Time

b) Product quality e)Project/product information

c) Product quantity

5) How does a software project manager need to act to minimize the risk of software failure? [ ]

a) Double the project team size

b) Request a large budget

c) Form a small software team

d) Track progress

e) Request for more period of time.

6) Which one of the following is a functional requirement [ ]

a) Maintainability

b) Portability

c) Robustness

d) None of the mentioned

7) The Software Requirement Specification(SRS) is said to be ________ if and only if no subset of individual requirements described in it conflict with each other. [ ]

a) Correct b) Consistent c) Unambiguous d) Verifiable

8) Which one of the following is NOT desired in a good software requirement specifications(SRS) document? [ ]

a) Functional requirements

b) Non-Functional requirements

c) Goals of implementations

d) Algorithm for software implementation.

9) When is the requirement specification activity carried out? [ ]

a) During requirements gathering activity

b) Before requirements analysis activity

c)Before requirements gathering activity

d) After requirements analysis activity

10) Which one of the following is not a requirements gathering technique?

a) Task analysis b) Scenario analysis [ ]

c) Form analysis d) SRS document review

SECTION-B

Descriptive Questions

1) Explain briefly **the value of good SRS and** the Requirements Engineering Process.
2) Give the Structure of Software Requirements Specification Document.
3) Design a SRS Document for Online Banking System?
4) Describe the Functional Specification Technique with use cases.
5) What is SRS? Discuss the characteristics of SRS.
6) Design a SRS Document for ATM System?
7) Design a SRS Document for Library Management System?

8) Briefly describe the functional specification with usecase with an example of auction system.

SECTION-C

C) Previous Gate Questions

1) Which one of the following is NOT desired in a good Software Requirement Specifications (SRS) document? **(GATE 2011)**

a) Functional Requirements

b) Non-Functional Requirements

c) Goals of Implementation

d) Algorithms for software implementation [ ]

2) A Software Requirements Specification (SRS) document should avoid discussing which one of the following? **(GATE 2015)**

a) User interface issues

b) Non-functional requirements

c) Design specification

d) Interfaces with third party software [ ]

3) Software requirement Specification(SRS) is also known as specification of: (**Nielit Scientist-2016**)

a) White box testing b) Grey box testing [ ]

c) Acceptance testing d) Black box testing

Unit – V

Software Architecture and Design

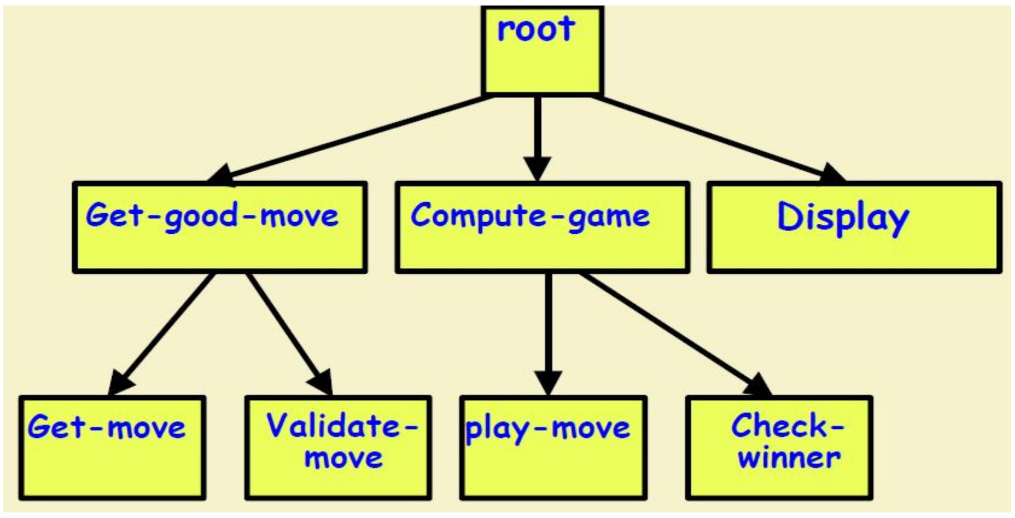
5.1 Role of Software Architecture

- What is achieved during design phase?
- Typically, we transform SRS document to design document. And the design can easily be implemented in some programming language.
- Design is usually classified into two types;

  1. High level design (also called Software Architecture)

  2. Detailed design
- Generally speaking, architecture of a system provides a very high level view of the parts of the system and how they are related to form the whole system.
- Formally we define software architecture as

  The software architecture of a system is the structure of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
- For example, consider the Tic-tac game. Its architecture can be shown below.



- Some of the important uses that software architecture play are;

1. **Understanding and Communication** – There are several stakeholders for a software system – users (who use the system),

clients (who pay for the system), analysts (who analyze the requirements of the system), designers (who design), testers (who test), document writers (who prepare user manuals) etc. Through the architecture, the stakeholders gain an understanding of the system. In addition, it acts as a vehicle for providing communication between the stakeholders.

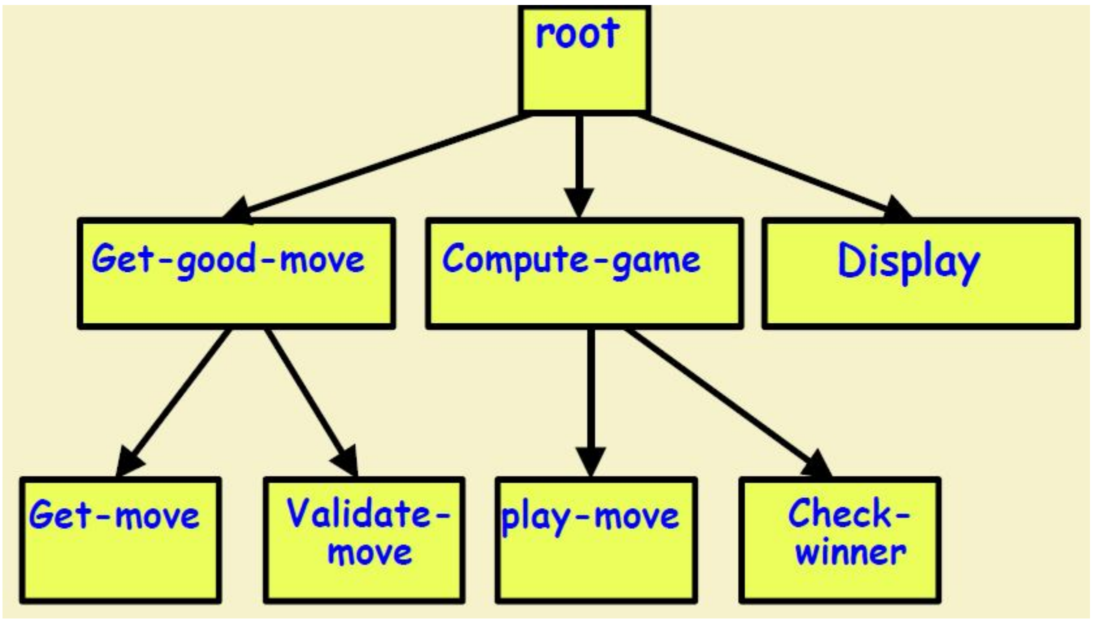
2. **Reuse** – For a long time, the software engineering world is been working toward a discipline where software can be assembled from parts that are developed by different people and are available for others to use. For example, the components like date, money, APIs etc are the reusable components which are used in many software systems. The architecture facilitates the analysts to decide which components can be reused and which can be developed. This is a crucial decision which speeds up the development of the software system.
3. **Construction and Evolution** – As we know that, the software architecture partitions the system into parts that are relatively independent to each other. This allows that different development teams to work on simultaneously on different parts.
4. **Analysis** – One of the important properties of the architecture is that it can be used to analyze the system before the system is actually built. It allows the designers to analyze the system in terms of its cost, reliability, performance, etc. For example, while building a website for shopping, it is possible to analyze the response time or throughput for a proposed architecture, given some assumptions about the request load and hardware. It can then be decided whether the performance is satisfactory or not, and if not, what new capabilities should be added (for example, a different architecture or a faster server for the back end) to improve it to a satisfactory level.

5.2. Architecture Views

- In general, there is no unique architecture of a system. For example, consider the construction of a multi-storied building. In order to construct it, several plans like floor plans, plumbing plan, electricity plan, fire safety plan, elevation plan etc are created. These drawings are not independent of each other - they are all about the same building. However, each drawing provides a different view of the building.
- Similarly, in software development also several plans are created. These different plans are called as views of the system. A view represents the system as composed of different components of the system and their relationships.
- Many types of views have been proposed. Most of the proposed views generally belong to one of these three types;
    1. Module view
    2. Component and Connector view
    3. Allocation view

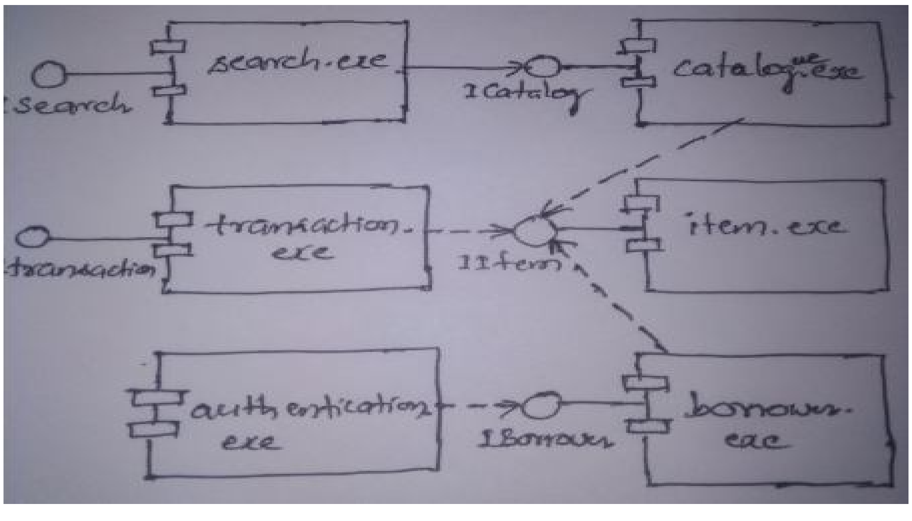
Module View

- In a module view, the system is viewed as a collection of modules and their relationships.
- Each module performs a well defined functionality. For example, the module view of Tic-tac software is shown below.



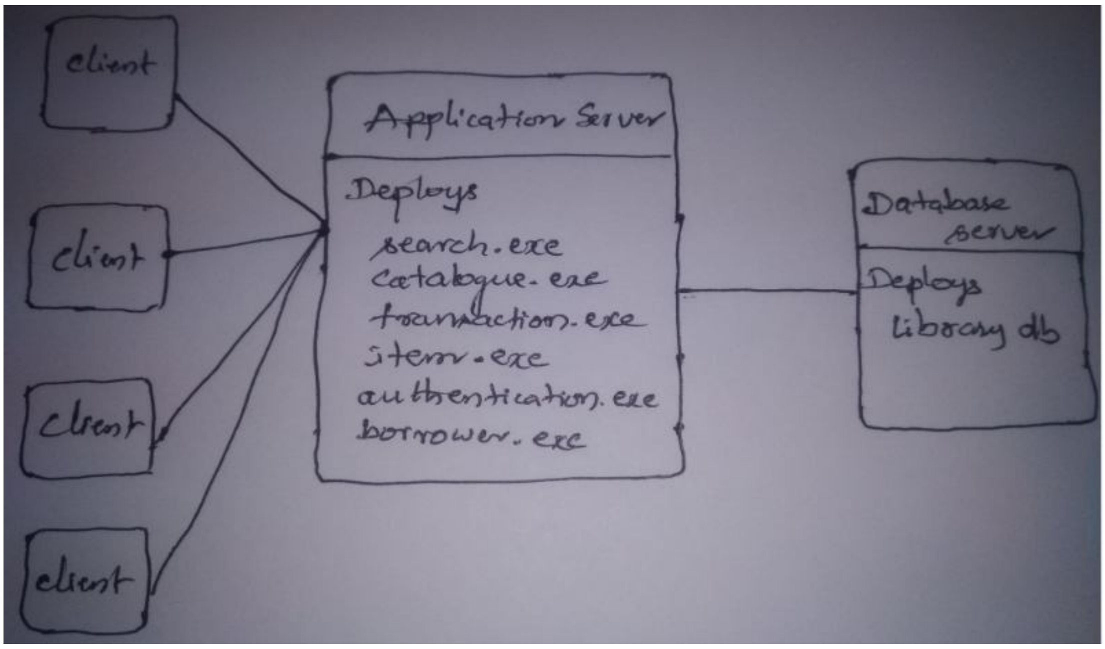
Component and Connector View

- In a component and connector (C&C) view, the system is viewed as a collection of runtime entities called components.
- A component is a unit which has an identity in the executing system. Objects (not classes), a collection of objects, and a process are examples of components.
- While executing, components need to interact with others to support the system services. Connectors provide means for this interaction. Examples of connectors are interfaces, pipes and sockets.
- For example, the component and connector view of library system is shown below.



Allocation View

- An allocation view focuses on how the different software units are allocated to resources like the hardware, file systems, and people.
- That is, an allocation view specifies the relationship between software elements and the environments in which the software system is executed.
- They expose structural properties like which processes run on which processor, and how the system files are organized on a file system.
- For example, the allocation view of library system is shown below.



5.5 Function Oriented Design

- In this, the system is looked upon as a set of functions.
- That is, each function specified in the SRS document is decomposed into more detailed functions.
- That is why it is also called as the top-down approach.
- Functions are then mapped to a module structure.
- For example, the function **create-new-library-member** can be decomposed into sub functions as
  - creates the record for a new member,
  - assigns a unique membership number

- prints a bill towards the membership

- Function oriented design can be graphically represented using a structure chart.

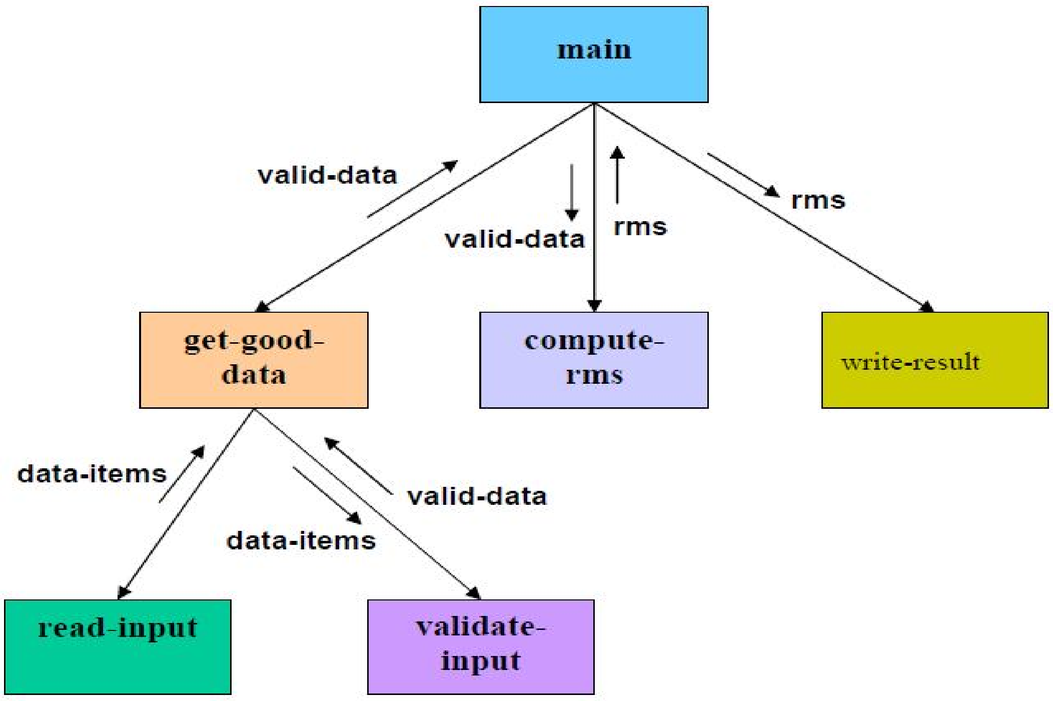
5.5.1 Structure Chart

- A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules.
- Hence, the structure chart representation can be easily implemented using some programming language.
- Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.
- The basic building blocks which are used to design structure charts are the following:
  - **Rectangular boxes:** Represents a module.
  - **Module invocation arrows:** Control is passed from one module to another module in the direction of the connecting arrow.
  - **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
  - Library modules: Represented by a rectangle with double edges.
  - **Selection:** Represented by a diamond symbol.
  - Repetition: Represented by a loop around the control flow arrow.
- For example, the following figure shows the structure chart for RMS (Root Mean Square) software. RMS is a kind of average used by statisticians and engineers. The RMS of a set of numbers can be computed as follows;
  - SQUARE all the values
  - Take the average of the squares
  - Take the square root of the average

Formula: $\text{RMS} = \sqrt{\frac{a_1^2 + a_2^2 + a_3^2 + \cdots + a_n^2}{n}}$ or $\sqrt{\frac{\sum_{i=1}^{n} a_i^2}{n}}$

Example: For the numbers 4 and 9,

$$\text{RMS} = \sqrt{\frac{4^2 + 9^2}{2}} \approx 6.96$$



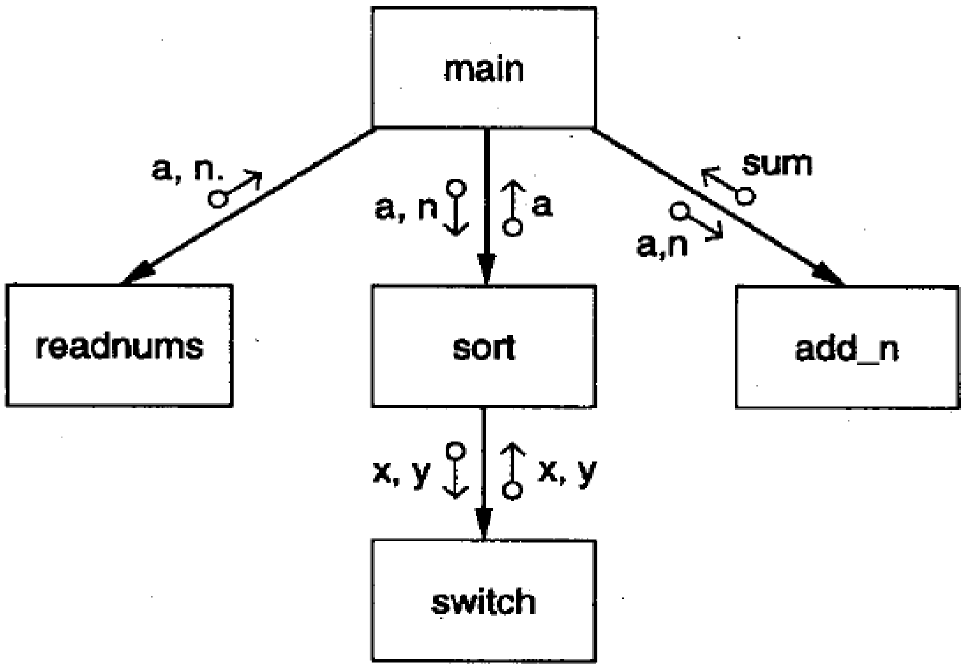
- In a structure chart, an arrow from module A to module B represents that module A invokes module B.
- As another example, consider the following program, whose structure chart is shown below.

```
main()
{
     int sum, n, N, a[MAX];
     readnums(a, &N); sort(a, N); scanf(&n);
     sum = add_n(a, n); printf(sum);
}

readnums(a, N)
int a[], *N;
{
     :
}

sort(a, N)
int a[], N;
{
     :
     if (a[i] > a[t]) switch(a[i], a[t]);
     :
}

/* Add the first n numbers of a */
add_n(a, n)
int a[], n;
{
     :
```



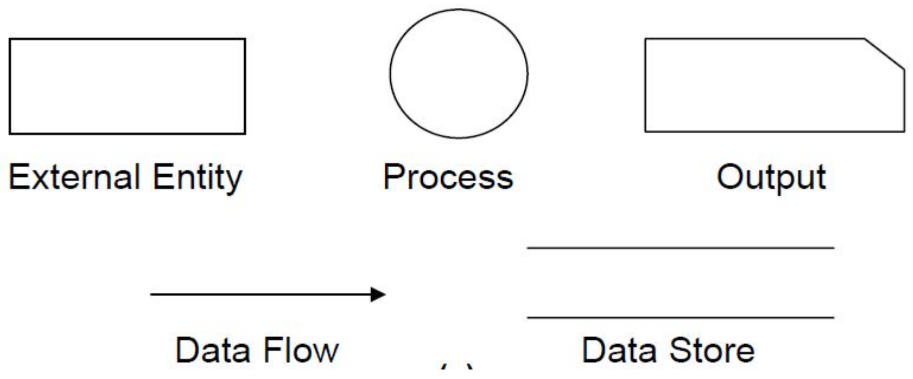
5.5.2 Structured Design Methodology

- The basic principle behind the structured design methodology is problem partitioning.

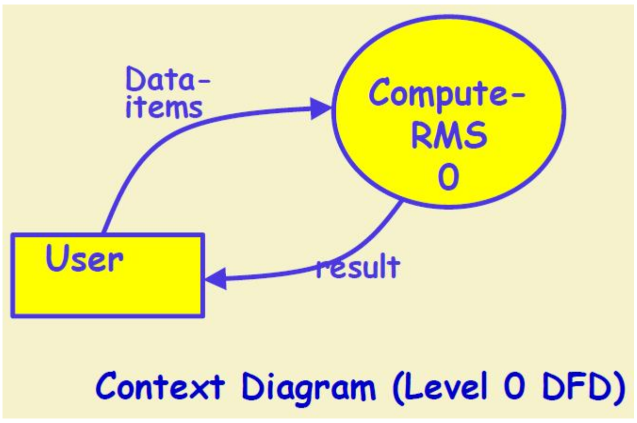
- That is, the system is decomposed into modules, modules are then decomposed into sub modules, sub modules are then decomposed into sub-sub modules and so on until the modules are small enough to handle them easily. This forms the hierarchy of modules.
- There are four major steps in the methodology.
    1. Restate the problem as a data flow diagram
    2. First-level factoring
    3. Identify the input and output data elements
    4. Factoring of input, output, and transform branches

1. Restate the problem as a data flow diagram

- To use this methodology, the first step is to construct the data flow diagram for the problem.
- The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data exchanged among these functions.
- Each function is considered as a processing station (or process) that consumes some input data and produces some output data.
- The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system.
- A DFD model uses a very limited number of primitive symbols as shown below.



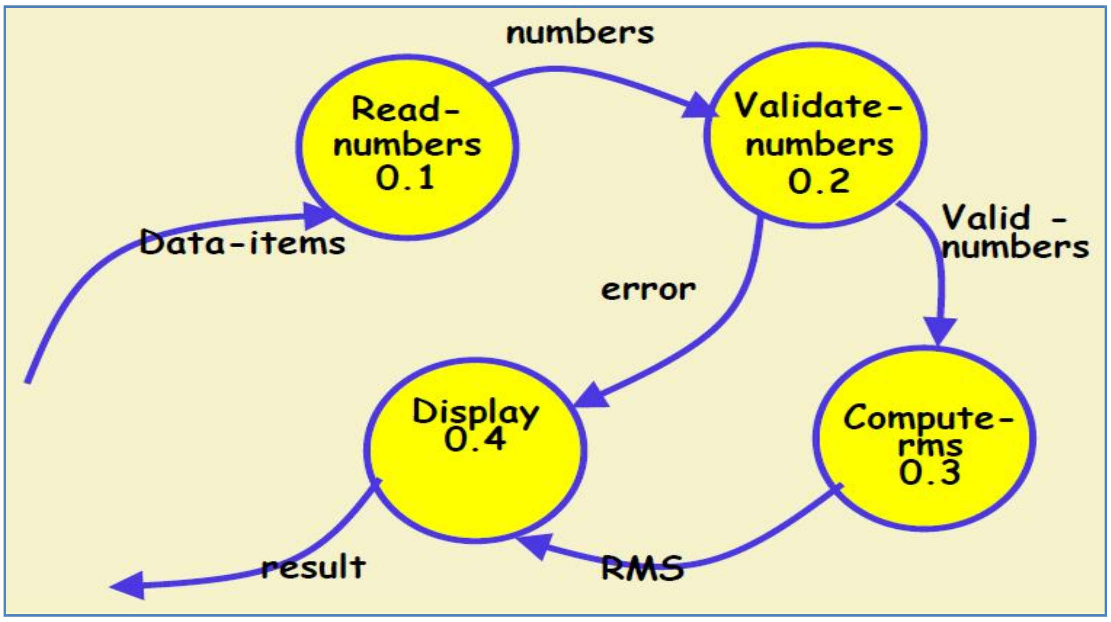
Example 1: RMS calculating software



The RMS calculating software would read three integer numbers from the user in the range of -1000 and +1000 and then determine the root mean square (RMS) of the three input numbers and display it. In this example, the context diagram (shown here) is simple to draw. The system accepts three integers from the user and returns the result to him.

2. First-level Factoring

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring a bubble. For example, in the above DFD *Compute-RMS* can be factored as shown below leading to level 1 DFD.



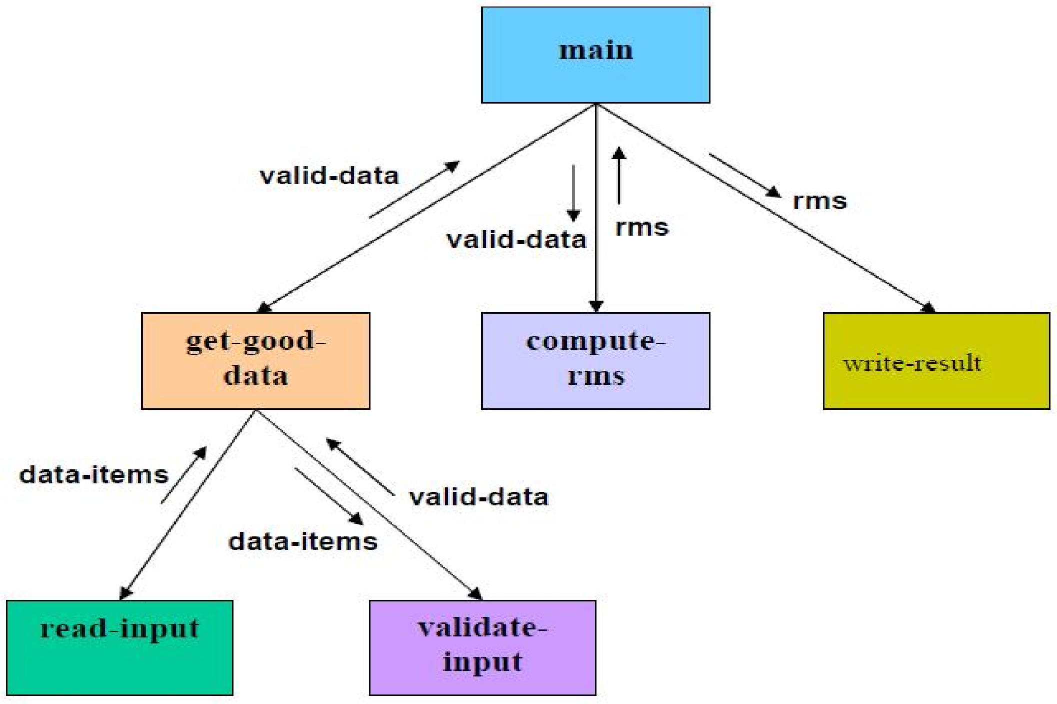
3. Identify the input and output data elements

- After factoring the DFD, the next step is to identify the input and output elements in the DFD.
- An input element in a DFD represents a process or processes which convert input data from physical to logical form. For example reading characters from the terminal and storing them in the internal table list is considered as an input element.
- In the above DFD, Read-numbers is the input element.
- Similarly, an output element in a DFD is one which transforms output data from logical form to physical form. Ex. From list or array into output characters.
- In the above DFD, Display is the output element.
- In addition to input and output elements it is also required to identify the transformation elements in the DFD. For example, in the above DFD Compute-rms is a transformation element.

4. Factoring of input, output, and transform branches

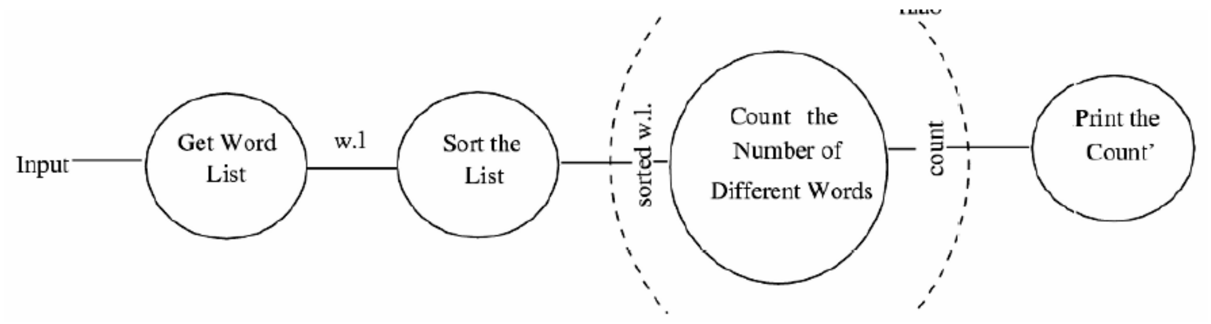
After identifying the input, output and transformation elements in the DFD the next step is to factor them further. That is, decompose the input element into sub input elements and so on.

The output of the structured design methodology is a structure chart. For example, the structure chart for Compute RMS software is shown below.



Example 2

Consider the problem of determining the number of different words in an input file. The data flow diagram for this problem is shown below.

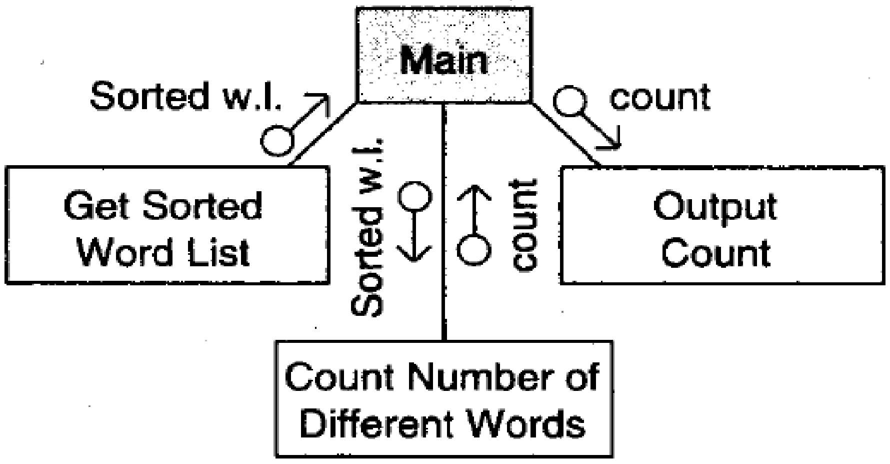


Input element – Get Word List

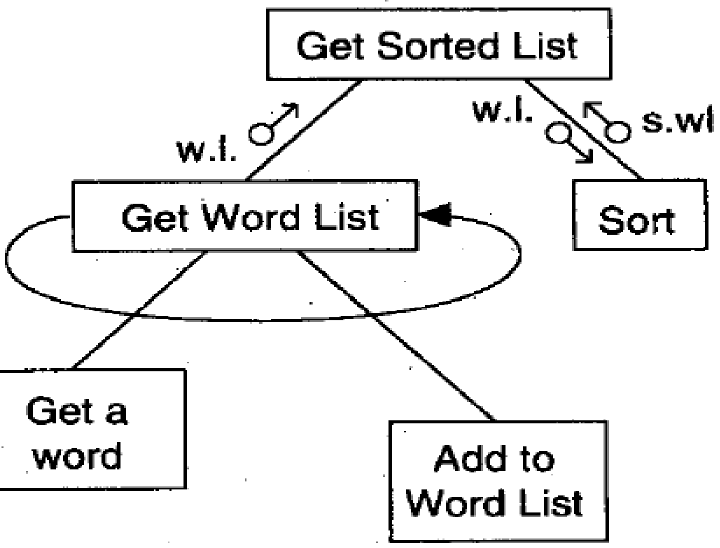
Output element – Print Count

Transformation Element(s) – Sort List and Count Number of Words

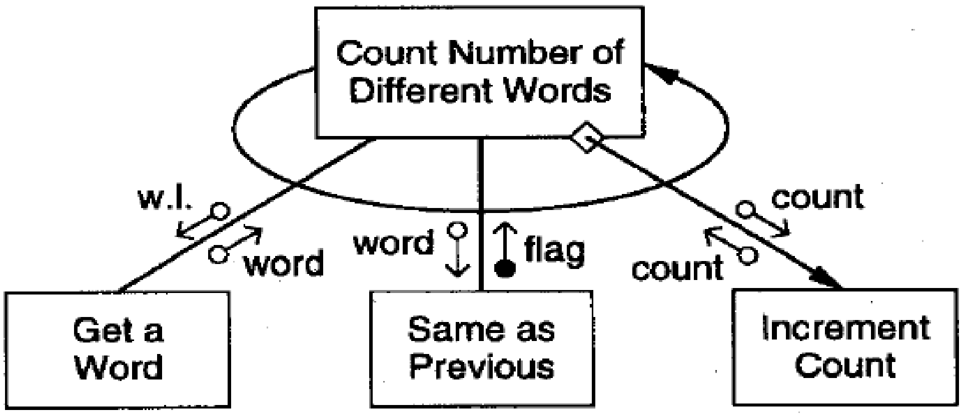
First level factoring



Factoring of Input module



Factoring of Transform module



5.6 Object-Oriented Design

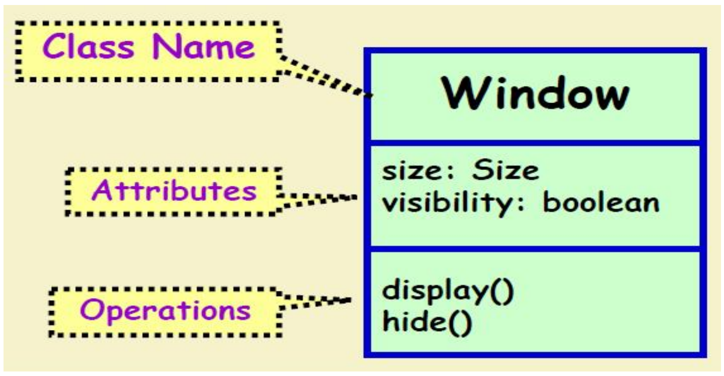
- There are typically two types of designs
    1. Function-oriented design
    2. Object-oriented design

Function-oriented design

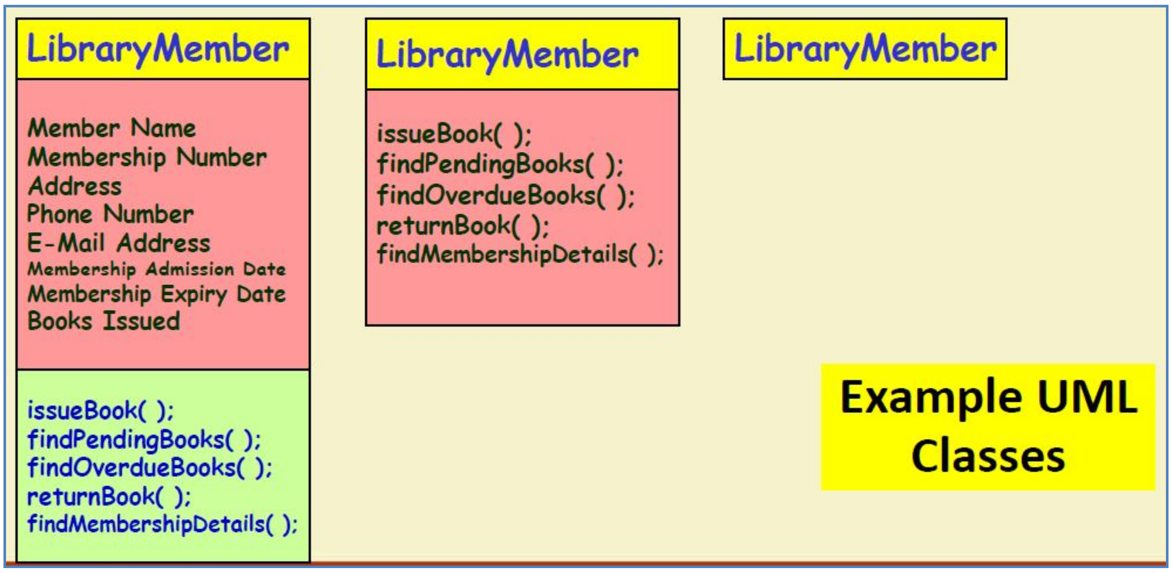
- It is the traditional design.
- In function-oriented design, the system is decomposed into set of functions called modules forming a hierarchical structure of the system.
- Here, the main building block is function or module.
- It follows top-down approach (i.e. the system is decomposed into modules, modules are then decomposed into sub modules, sub modules are then decomposed into sub-sub modules and son).
- Each module can be coded in a procedural language like C.

Object-oriented design

- It is the modern style of design.
- In object-oriented design, the main building block is an object or a class.
- It follows bottom-up approach (i.e. we start with the objects, objects are then combined into components, components are then combined to form the entire system).
- In this, the system is coded in an object oriented language like Java.
- An object is an entity or thing in the real world. For example, in Library Management system we could find set of objects such as Student, Book, Professor etc.
- A class is a set of objects that share the same attributes and operations.
- Object oriented design is best expressed in UML (Unified Modelling Language).
- In UML, a class is represented as a rectangle with three compartments as shown below.



- The following figure shows alternative representations of a class.



- Object-oriented design is concerned with finding and describing the software objects and how they collaborate to full fill the requirements.
- For example, in Library Management system the *Student* object collaborates with the *Book* object to full fill the requirement *Issue book*.

Example 1

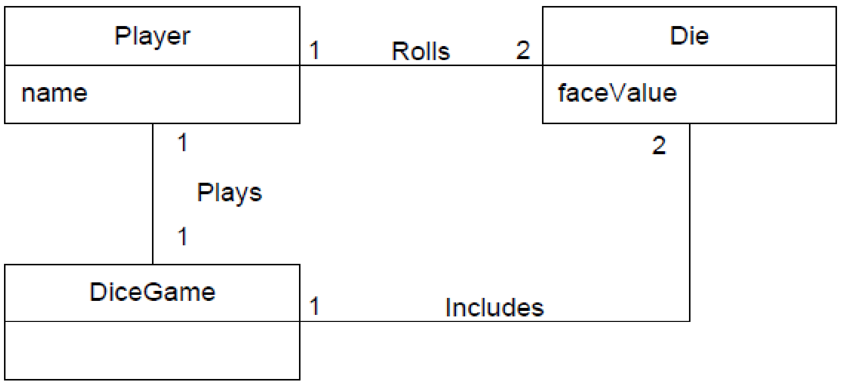
Let us we discuss the object-oriented design using a simple example - Dice game example in which a player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.

After investigating this problem, we can find the following classes to solve the problem.

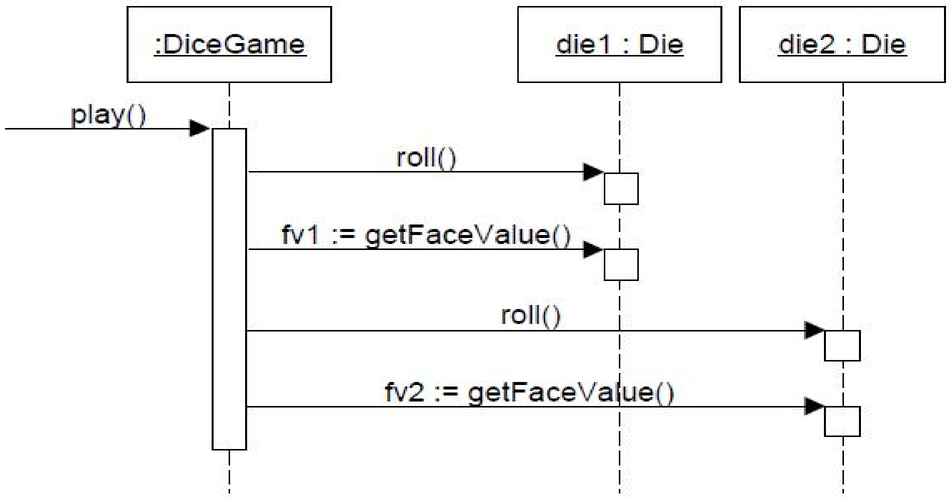
- Player

- DiceGame and Die

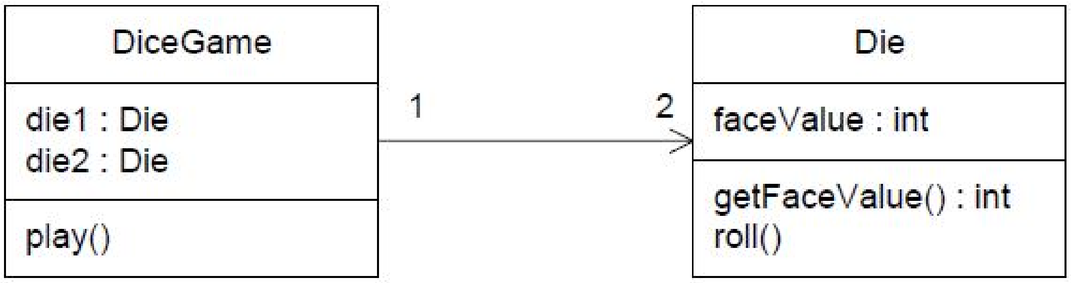
With these classes, now we can able to create the initial class diagram as shown below.



In order to find the operations of these classes, we need to create the interaction diagram for the Dice Game. The interaction diagram is shown below.



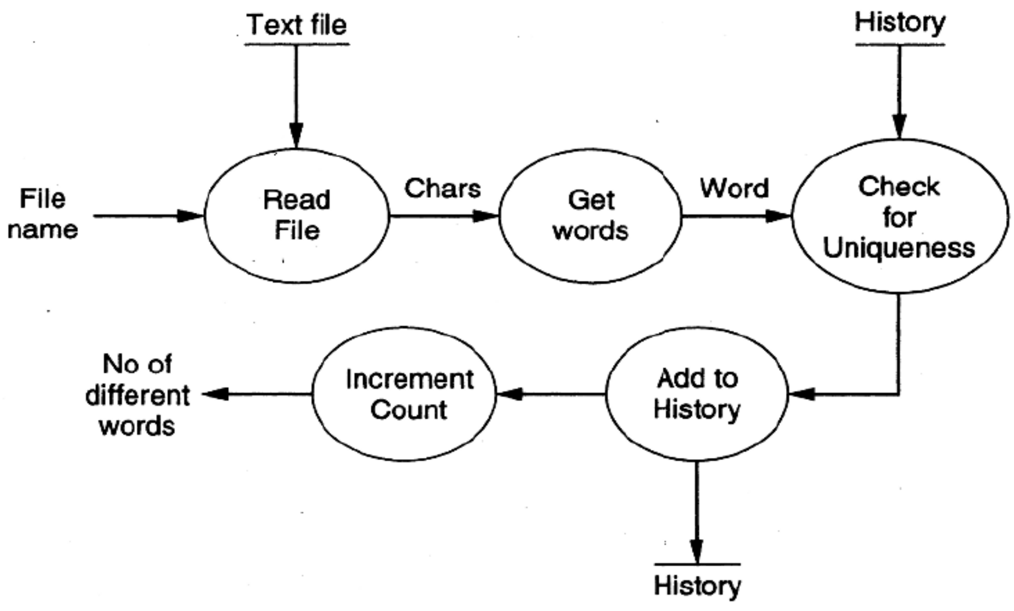
Notice that although in the real world a player rolls the dice, in the software design the DiceGame object "rolls" the dice. An inspection of the interaction diagram leads to the final class diagram shown below. Since a *play* message is sent to a *DiceGame* object, the *DiceGame* class requires a *play* method, while class *Die* requires a *roll* and *getFaceValue* method.



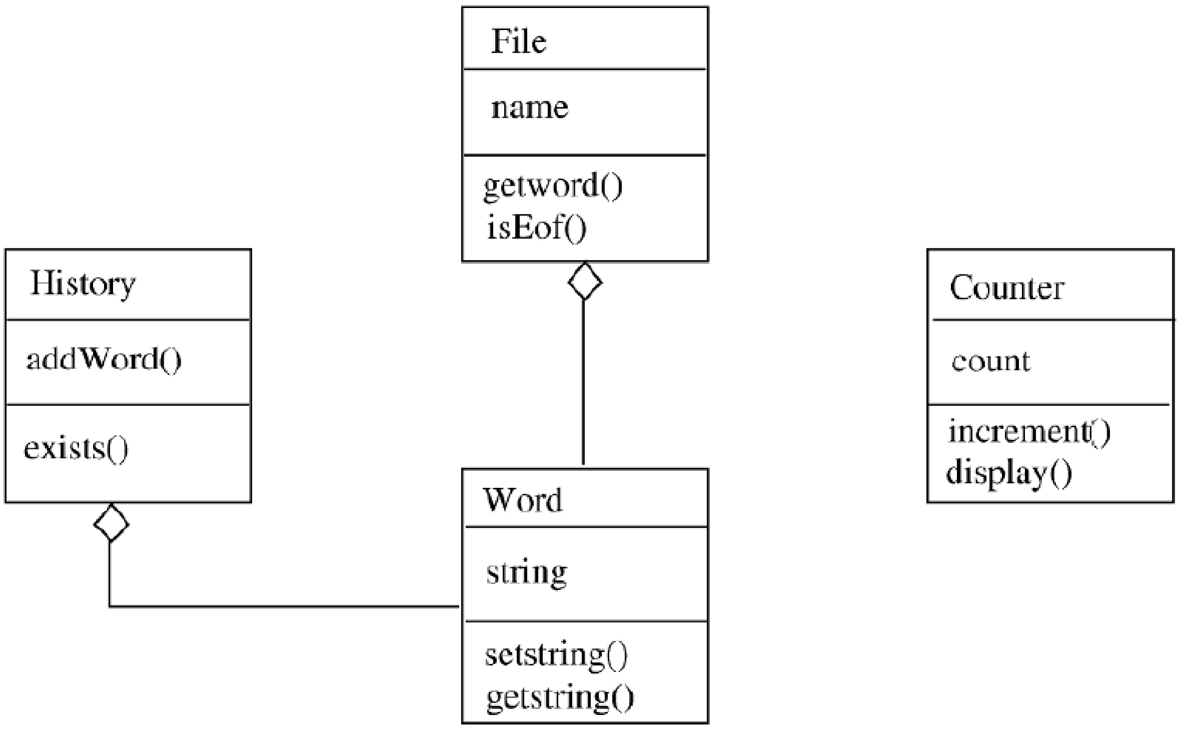
Example 2

The word counting problem - determine the number of different words in an input file.

The functional model of the problem is shown below.



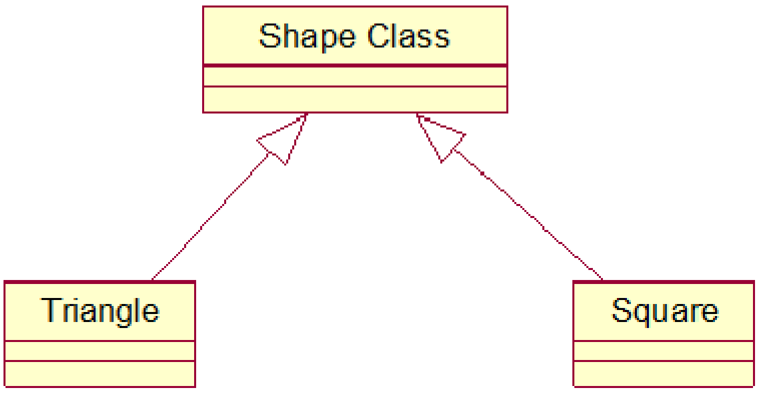
The class diagram for this problem is shown below.


File
name
getword()
isEof()
History
addWord()
exists()
Counter
count
increment()
display()
Word
string
setstring()
getstring()

UNIT V
Assignment-Cum-Tutorial Questions

Objective Questions

1) A component model defines standards for [ ]
   a) Properties b) Methods
   c) Mechanisms d) All of the mentioned
2) What makes a good architecture? [ ]
   a) The architecture may not be the product of a single architect or a small group
   b) The architect should have the technical requirements for the system and an articulated and prioritized list of qualitative properties
   c) The architecture may not be well documented
   d) All of the mentioned
3) To capture and access data from the store by various components we use [ ]
   a) Component Connector Structure
   b) Work-Allocation of Modules
   c) Component and Hardware Dependency
   d) Module Dependency Structure
4) Identify the architectural style which is most frequently used as web system backend [ ]
   a) Client Server Architectural Style
   b) Shared Data Style
   c) Peer to-Peer Style / Object Oriented Style
   d) Publish-Subscribe Style
5) Select the architectural style which is used for Events like Mouse Clicking, mouse drag and database events etc [ ]
   a) Peer-to-Peer Style
   b) Client server Style
   c) shared Data style
   d) Publish-Subscribe Style
6) Which of the following can be considered regarding client and server? [ ]
   a) Client and Server is an architectural style
   b) Client and Server may be considered as an architectural style
   c) Client and Server is not an architectural style
   d) None of the mentioned
7) Choose the option that does not define Function Oriented Software Design [ ]
   a) It consists of module definitions
   b) Modules represent data abstraction
   c) Modules support functional abstraction
   d) None of the above
8) What type of relationship is represented by Shape class and Square ? [ ]



a) Realization b) Generalization
c) Aggregation d) Dependency

9) Which diagram in UML shows a complete or partial view of the structure of a modelled system at a specific time? [ ]

a) Sequence Diagram b) Collaboration Diagram
c) Class Diagram d) Object Diagram

10) Which design defines the logical structure of each module and their interfaces that is used to communicate with other modules. [ ]

a) High level design b) Architectural Design
c) Detailed design d) All mentioned above

SECTION-B

Descriptive Questions

1) What is a Software Architecture? Explain important uses of software architecture?
2) Write about the Architecture views in detail.
3) Briefly explain about Architecture Styles in detail.
4) Identify first level factoring activities for design methodology and apply that for ATM.
5) Differentiate the Component and Connector views.
6) Illustrate architecture diagram for Student Survey System.
7) Apply the suitable style for course scheduling.
8) Illustrate the Authentication and cache management in the Architecture of survey system.
9) Illustrate structure chart of the sort program for
    a. Representation of different types of Modules.
    b. Iteration and decision representation.

10) What are the metrics that can be used to study complexity of an object-oriented design.
11) Draw DFDs for
    a. ATM and
    b. Word Count problem.

C) Previous GATE/UGC NET Questions:

1) ______________________________ of a system is the structure or structures of the system which comprise software elements, the externally visible properties of these elements and the relationship amongst them. **[UGC NET JUNE 2013]**

a) Software construction	b) Software evolution
c) **Software architecture**	d) Software reuse

*****

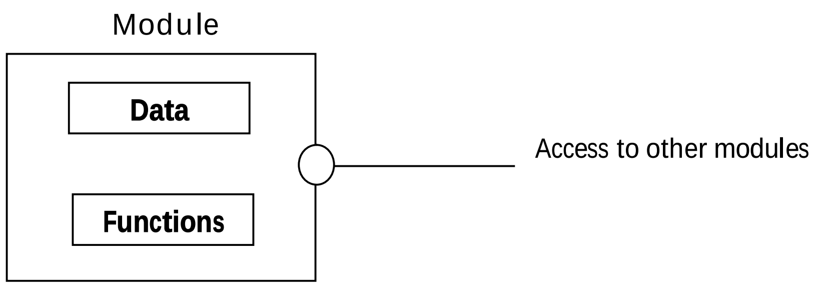
Coding and Testing

6.1 Programming Principles and Guidelines

- The software engineer translates the design into source code.
- The main goal of programming is to produce quality source codes that can reduce the cost of testing and maintenance.
- A clear, readable and understandable source code will make other tasks easier.
- The quality of source code depends on the skills and expertise of software engineers.
- The following programming principles can make a code clear, readable and understandable.
    - Information hiding
    - Structured programming
    - Coding standards

6.1.1 Information Hiding

- Information or data hiding is a programming concept which protects the data from direct modification by other parts of the program.
- It can be achieved using data encapsulation.
- Binding the data and operations that operate on the data into a single unit is called as data encapsulation. This is shown below.

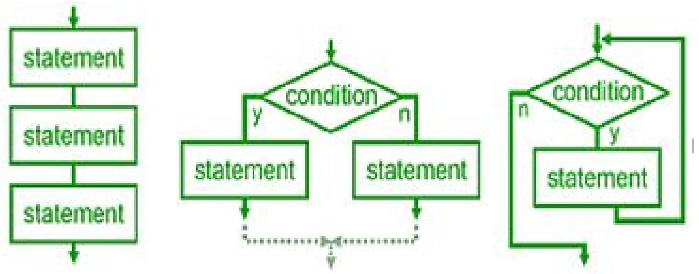


- The above figure shows that the functions within the module can only access the data in the module. Functions outside the module cannot access the data because the data is not visible to these functions. However, the module may provide an interface through which other functions can access the data present in the module.

- In java, information hiding can be achieved by an object or class.

6.1.2 Structured Programming

- The basic objective of coding activity is to produce programs that are easy to understand.
- Structured programming helps to develop programs that are easier to understand.
- Every structured programming consists of the following three features
    - Sequence – statements in a program are executed sequentially one after another.
    - Selection – based on the result of the condition, statements are executed.
    - Iteration – based on the condition, group of statements are executed repeatedly.
- The following figure shows these features.



Sequence Selection Iteration

6.1.3 Coding Standards

- Programmers spend considerable amount of time reading the code during debugging and enhancement.
- Therefore, it is of prime importance to write code in a manner that is easy to read and understand.

- Coding standards provide rules and guidelines in order to make code easier to read.
- Most organizations that develop software regularly develop their own standards.
- In general, coding standards provide guidelines for programmers regarding naming, file organization, statements and declarations, and comments.

Naming

- Package names should be in lowercase (e.g., mypackage, edu.iitk.maths).
- Type names should be nouns and should start with uppercase (e.g., Day, DateOfBirth, EventHandler).
- Variable names should be nouns starting with lowercase (e.g., name, amount).
- Constant names should be all uppercase (e.g., PI, MAX ITERATIONS).
- Method names should be verbs starting with lowercase
- Loop iterators should be named i, j, k, etc.
- The prefix is should be used for Boolean variables and methods
- Exception classes should be suffixed with Exception (e.g., OutOfBoundException).

Files

- Java source files should have the extension .java
- File name should be same as the outer class name

Statements

- Variables should be initialized where declared
- Declare related variables together in a common statement.
- Class variables should never be declared public.
- Avoid the use of break and continue in a loop.

Comments

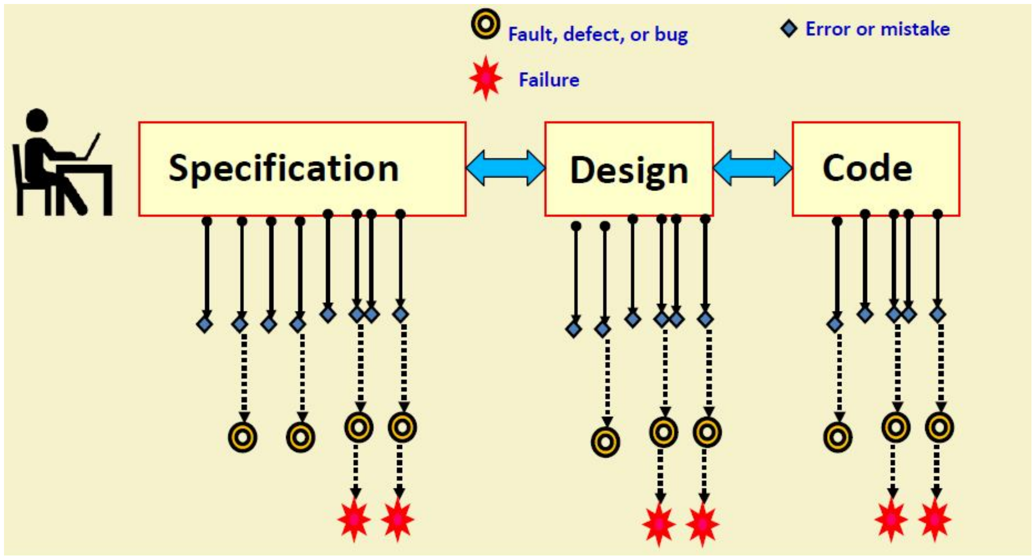
- Single line comments for a block of code.
- There should be comments for all major variables explaining what they represent.

6.2 Testing Concepts

- Testing is the process of executing a program with the intent of finding errors.
- In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

6.2.1 Error, Fault, and Failure

- **Error** - People make errors. Errors are also called as mistakes. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.
- **Fault** - is the representation of an error. That is errors may lead to faults. Fault is also called as defect or bug. Not all errors lead to bugs.
- **Failure** - A failure occurs when a fault executes.
- The following figure shows the relationship among error, fault and failure. It shows that some of the errors may lead to faults and some of the faults may lead to failures.



- For example consider the statement: **x<=50**. But, the programmer mistakenly written it as **x<=500**. Assuming that x never gets a value above 50. In such situation, though it is an error but, it will not lead to a fault.

6.2.2 Test Case, Test Suite and Test Harness

- A program is tested using a set of carefully designed test cases.
- **Test Case** - A test case is a triplet [I,S,O]
  - Where I is the data to be input to the system,
  - S is the state of the system at which the data will be input,
  - O is the expected output of the system.
- For example, to test the functionality Renew Book in Library Management System, the test case would be
  - **Set the program in the required state:** Book record created, member record created, Book issued
  - **Give the defined input:** Select renew book option and request renew for a further 2 week period.
  - **Observe the output**: Compare it to the expected output.
- **Test Suite** - The set of all test cases is called the test suite

Why design test cases?

- Exhaustive testing of any non-trivial system is impractical because, input data domain is extremely large.
- Therefore design an optimal test suite of reasonable size that uncovers as many errors as possible.
- Consider the following example. The code has a simple programming error:

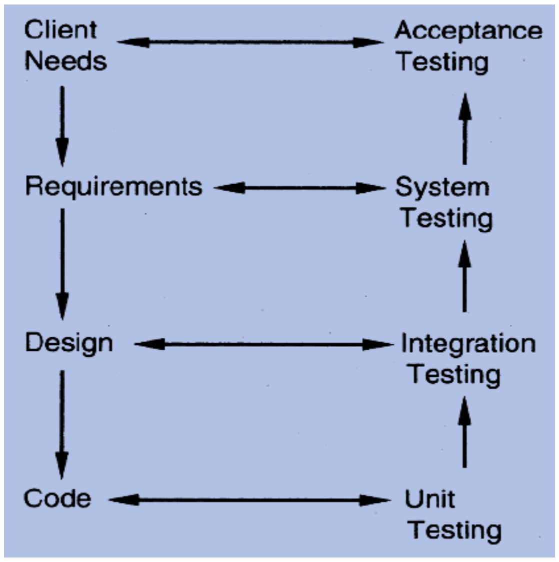
```
If (x>y) max = x;
    else max = x; // should be max=y;
```

- Test suite {(x=3,y=2);(x=2,y=3)} can detect the bug
- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)}does not detect the bug.
- **Test Harness**
  - To have a test suite executed automatically, we will need a framework in which test inputs can be defined representing test cases, the automated test script can be written, the SUT can be executed by this script, and the result of entire testing reported to the tester.

- Many testing frameworks now exist that permit all this to be done in a simple manner.
- A testing framework is called as **test harness**

6.2.3 Levels of Testing

- Software is tested at 4 levels of testing.
  1. Unit testing
  2. Integration testing
  3. System testing and
  4. Acceptance testing
- **Unit testing** - Test each module (unit, or component) independently. Mostly done by the developers of the modules.
- **Integration testing** - many unit tested modules are combined into subsystems, which are then tested. Here, the emphasis is on testing the interfaces between the modules.
- **System testing and Acceptance testing**
  - Here the entire software system is tested.
  - The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements.
  - Acceptance testing is often performed by the clients or customers to demonstrate that the software is working satisfactorily.
- The following figure shows levels of testing.



6.3 Testing Process

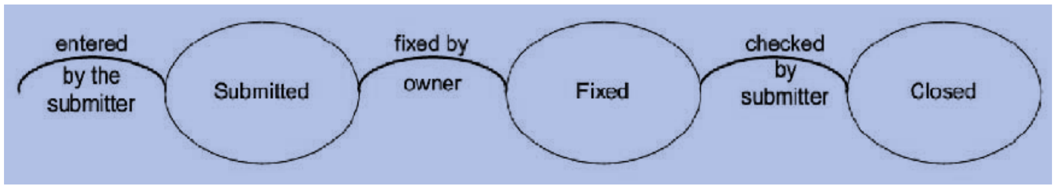
- The testing process for a project consists of three high-level tasks.
    1. Test planning
    2. Test case design and
    3. Test execution

1. **Test Plan -** Testing commences with a test plan. A test plan should contain the following:
    - Test unit specification
    - Features to be tested
    - Approach for testing
    - Test deliverables
    - Schedule and task allocation

- *Test unit specification* – A test unit is a set of one or more modules that form software under test (SUT).
- *Features to be tested* - include all software features and combinations of features that should be tested. They typically include functional requirements, performance requirements and design constraints.
- *Approach for testing* – specifies testing type for executing the test cases. It can be a block box or white box testing.

- *Test deliverables* - Could be a list of test cases that were used, detailed results of testing including the list of defects found, test summary report, and data about the code coverage.
- *Schedule and Task allocation* – specifies schedule for doing testing and the persons who are responsible for doing the testing.

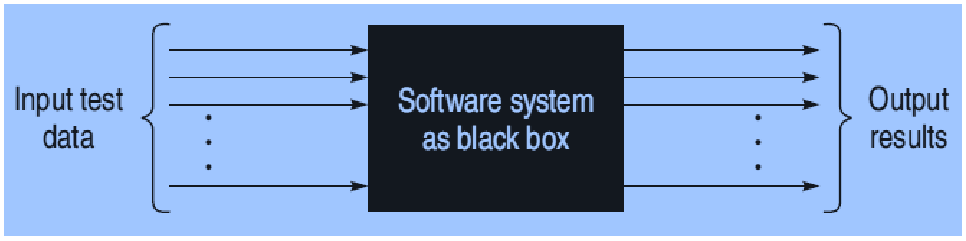
2. **Test Case Design** - In this set of test cases are designed for performing the testing. Test cases can be designed using black box or white box testing techniques.
3. **Test Case Execution** - With the specification of test cases, the next step in the testing process is to execute them. Testing tools can be used to execute test cases. During test case execution, defects are found. These defects are then fixed and testing is done again to verify the fix. The following figure shows the life cycle of a defect.



The figure shows that the defects found during the testing are submitted to the module developer at which stage the defect is said to be in *Submitted* state. Then, the developer debugs the program and fixes the bug in the program. At this time the defect is said to be in *Fixed* state. After this, the tester again tests the program to see whether bug fixation is over. If it is over then the bug is said to be in the *Closed* state.

6.4 Black-box Testing

- In black-box testing technique, test cases are designed based on functional specifications.
- Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software.
- In black-box testing, the tester has no idea of the internal structure of the software, so, it is called as black-box testing.



- Since, we test the functionalities of the software, so, black-box testing is also called as functional testing.
- Number of testing strategies is used to design the test cases in black-box testing.
  - Scenario coverage
  - **Boundary value analysis**
  - **Equivalence class partitioning**
  - **Cause-effect (Decision Table) testing**
  - Combinatorial testing
  - Orthogonal array testing

6.4.1 Boundary Value Analysis (BVA)

- Some typical programming errors occur at the boundaries of input values. Programmers often commit mistakes at the boundaries of input values.
- Programmers may improperly use <instead of <=
- Boundary value analysis selects test cases at the boundaries of input values. There are two variations in BVA.
  - Boundary Value Checking (BVC)
  - Robust Testing Method

Boundary Value Checking (BVC)

- In this, test cases are designed based on

  (a) Minimum value (Min)

  (b) Value just above the minimum value (Min+)

  (c) Maximum value (Max)

  (d) Value just below the maximum value (Max−)

  (e) Nominal value
- For example, for n variables, 4n+1 test cases can be designed

Example 1: **let there is a variable x with range [-3, 10]**

Test cases: **-3, -2, 10, 9, 0**

Example 1: let there is a variable y with range [5, 100]

Test cases: **5, 6, 100, 99, 50**

Robust Testing Method

- In this, test cases are designed based on

  (a) Minimum value (Min)

  (b) A value just less than Minimum value (Min−)

  (c) Value just above the minimum value (Min+)

  (d) Maximum value (Max)

  (e) Value just below the maximum value (Max−)

  (f) A value just greater than the Maximum value (Max+)

  (g) Nominal value
- In this, for n variables, 6n+1 test cases can be designed

  Example 1: **let there is a variable x with range [-3, 10]**

  Test cases: **-3, -4, -2, 10, 9, 11, 0**

  Example 1: let there is a variable y with range [5, 100]

  Test cases: **5, 4, 6, 100, 99, 101, 50**

6.4.2 Equivalence Class Partitioning

- The input values to a program are partitioned into equivalence classes
- Few guidelines for determining the equivalence classes can be given as
  - If an input is a range, one valid and two invalid equivalence classes are defined. Example: 1 to 100
  - If an input is a set, one valid and one invalid equivalence classes are defined. Example: {a, b, c}
  - If an input is a Boolean value, one valid and one invalid class are defined.

Example 1

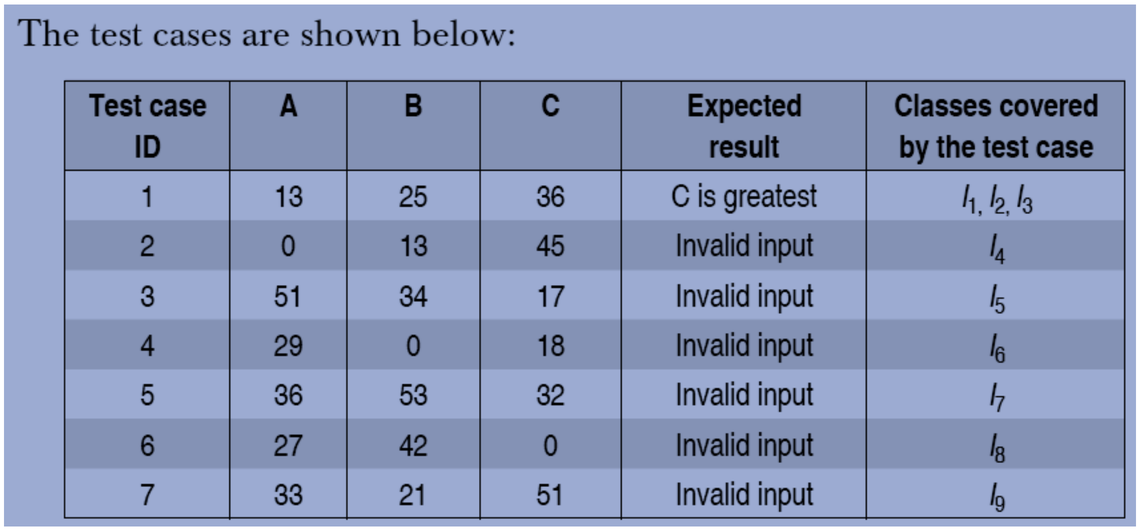
A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class partitioning testing technique.

Solution

First we partition the domain of input as valid input values and invalid values, getting the following classes:

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

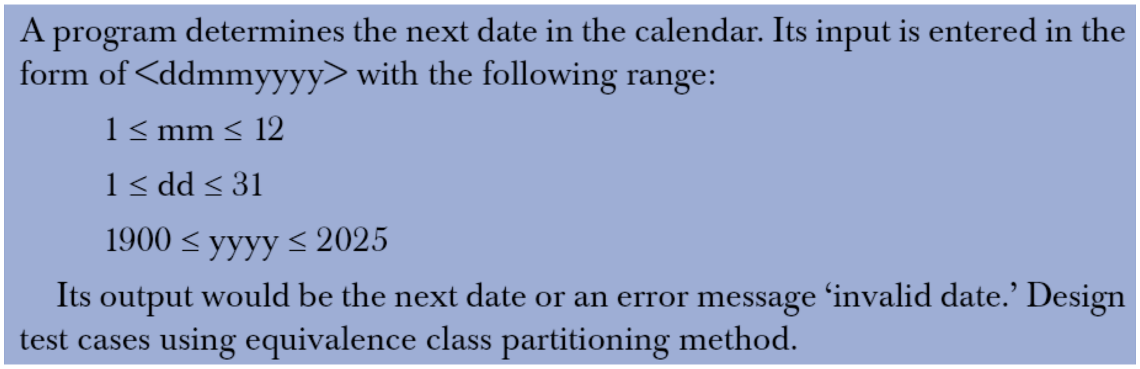
Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. This is shown below.



The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

Example 2



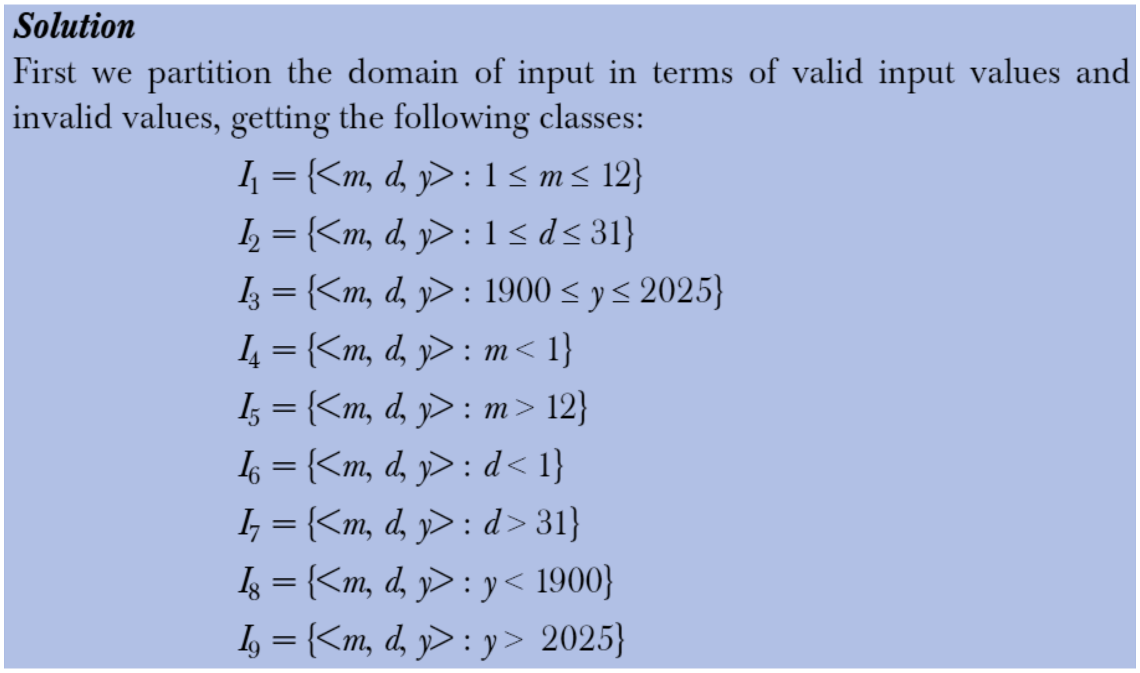
A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

$1 \leq \text{mm} \leq 12$

$1 \leq \text{dd} \leq 31$

$1900 \leq \text{yyyy} \leq 2025$

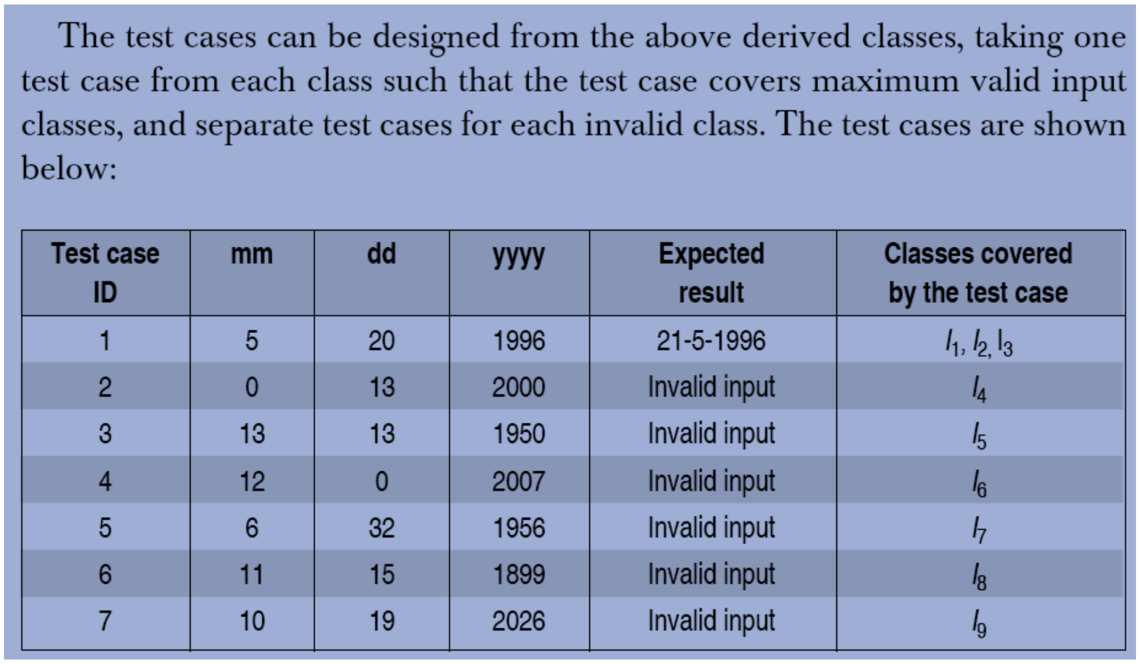
Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.



Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

$$I_1 = \{<m, d, y> : 1 \leq m \leq 12\}$$
$$I_2 = \{<m, d, y> : 1 \leq d \leq 31\}$$
$$I_3 = \{<m, d, y> : 1900 \leq y \leq 2025\}$$
$$I_4 = \{<m, d, y> : m < 1\}$$
$$I_5 = \{<m, d, y> : m > 12\}$$
$$I_6 = \{<m, d, y> : d < 1\}$$
$$I_7 = \{<m, d, y> : d > 31\}$$
$$I_8 = \{<m, d, y> : y < 1900\}$$
$$I_9 = \{<m, d, y> : y > 2025\}$$

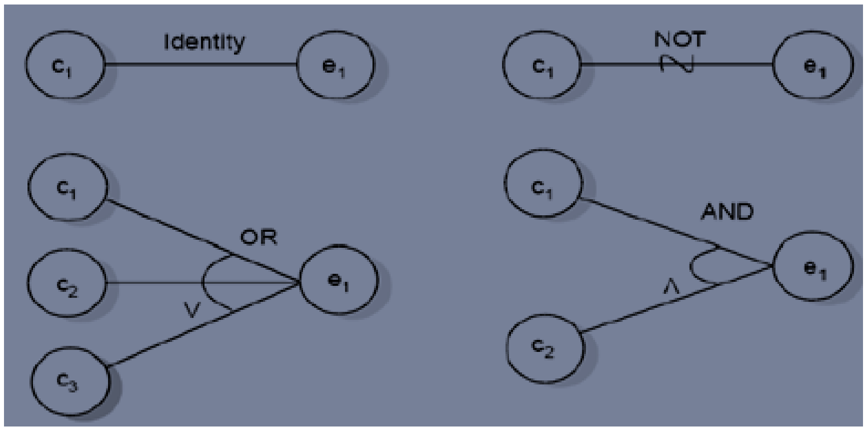


The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

6.4.3 Cause Effect Graphing Technique

- Causes & effects in the specifications are identified.
  - ❑ A cause is a distinct input condition.
  - ❑ An effect is an output condition or a system transformation.
- The semantic content of the specification is analysed and transformed into a Boolean graph linking the causes & effects.
- The graph is then converted to a decision table. Each row in the table represents a test case.
- The basic notations of the graph are shown below.

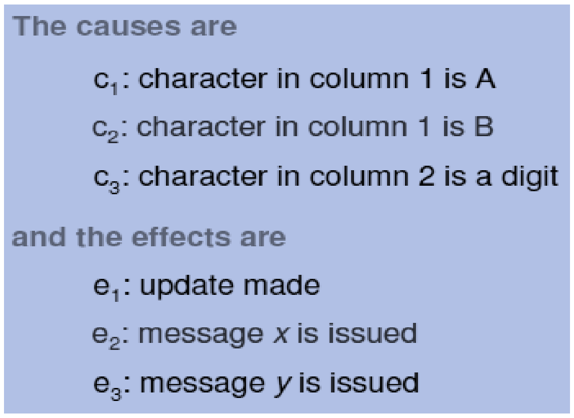


Example 1

The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message *x is issued. If* the character in column 2 is not a digit, message y is issued. Design the cause-effect graph for this scenario.

Solution

First identify the causes and effects in the problem statement.


The causes are

c_1: character in column 1 is A

c_2: character in column 1 is B

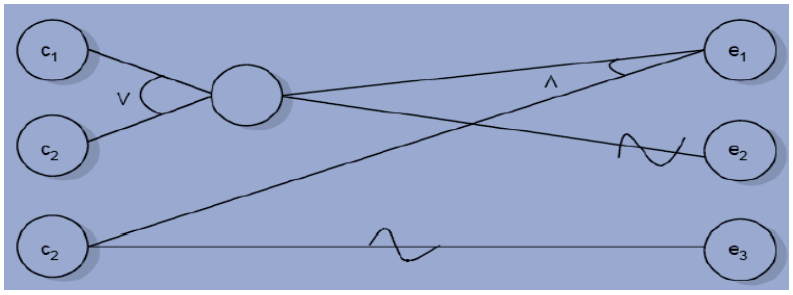
c_3: character in column 2 is a digit

and the effects are

e_1: update made

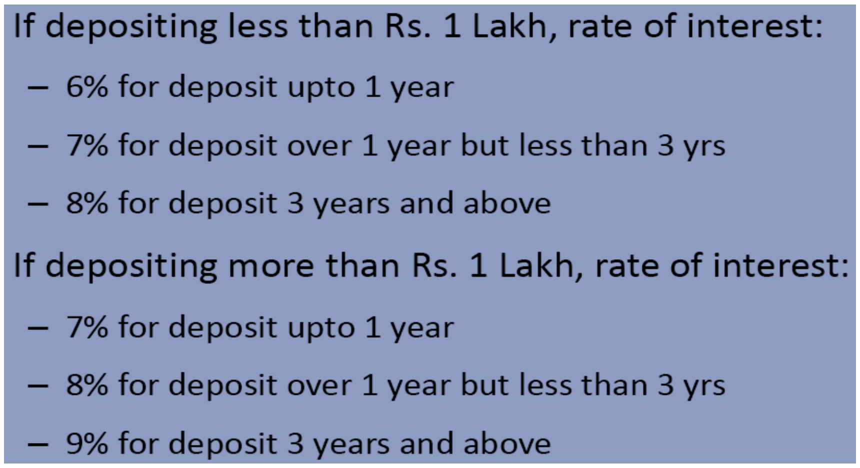
e_2: message *x* is issued

e_3: message *y* is issued



Example 2

Consider the following scenario.



If depositing less than Rs. 1 Lakh, rate of interest:

- 6% for deposit upto 1 year
- 7% for deposit over 1 year but less than 3 yrs
- 8% for deposit 3 years and above

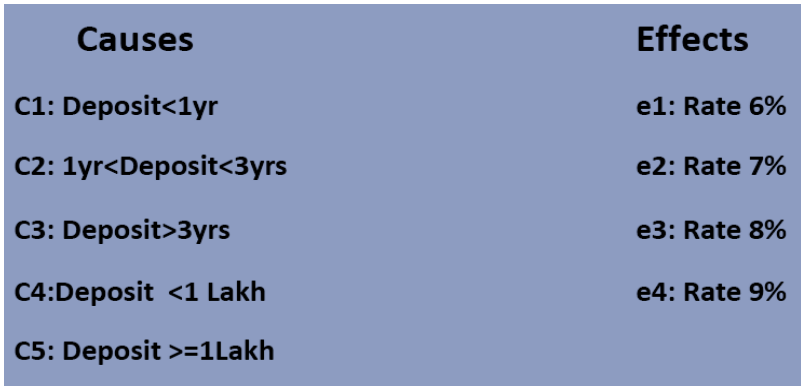
If depositing more than Rs. 1 Lakh, rate of interest:

- 7% for deposit upto 1 year
- 8% for deposit over 1 year but less than 3 yrs
- 9% for deposit 3 years and above

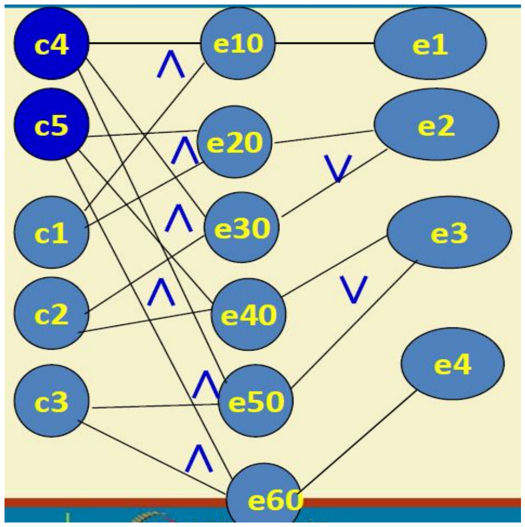
Design the test cases for this scenario using cause-effect graphing technique.

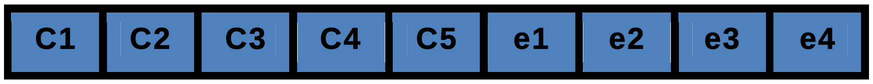
Solution

Identify the causes and effects and draw the graph.



Causes	Effects
C1: Deposit<1yr	e1: Rate 6%
C2: 1yr<Deposit<3yrs	e2: Rate 7%
C3: Deposit>3yrs	e3: Rate 8%
C4:Deposit <1 Lakh	e4: Rate 9%
C5: Deposit >=1Lakh	





C1	C2	C3	C4	C5	e1	e2	e3	e4

1	0	0	1	0	1	0	0	0
1	0	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	0	0	1	0	0	1	0
0	0	1	1	0	0	0	1	0
0	0	1	0	1	0	0	0	1

Each row in the table is considered as a test case.

6.5 White-box Testing

- In white box testing, test cases are designed based on the code structure of the program. Because of this it is also called as structural testing.
- In this, test cases are designed to cover the entire program.
- Several white-box testing strategies have become very popular.
    - Statement coverage
    - Branch coverage
    - **Path coverage**
    - Condition coverage
    - MC/DC coverage
    - **Mutation testing**
    - **Data flow-based testing**

6.5.1 Control Flow Testing (or Path Testing)

- In this, design test cases such that:
    - ✓ All linearly independent paths in the program are executed at least once.
- Defined in terms of Control flow graph (CFG) of a program.
- To understand the path coverage based testing: We need to learn how to draw control flow graph of a program.

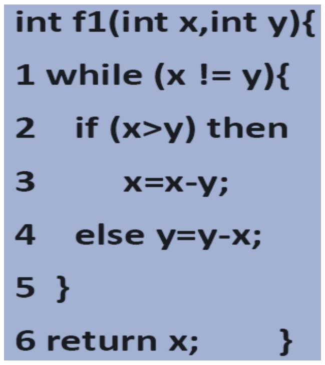
- A control flow graph (CFG) describes: The sequence in which different instructions of a program get executed and the way control flows through the program.

How to Draw Control Flow Graph?

- Number all statements of a program.
- Numbered statements: Represent nodes of control flow graph.
- Draw an edge from one node to another node: If execution of the statement representing the first node can result in transfer of control to the other node.

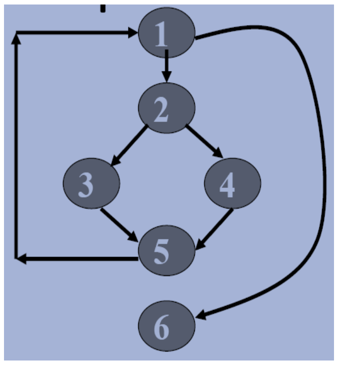
Example

Consider the following program.



```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```

Program



Control-flow graph

Once the control-flow graph is drawn, then we can find the number of independent paths in it. The number of independent paths present in the above graph is 3 as given below.

- ✓ **1,6** test case (**x=1, y=1**)
- ✓ **1,2,3,5,1,6** test case(**x=1, y=2**)
- ✓ **1,2,4,5,1,6** test case(**x=2, y=1**)

McCab's Cyclomatic Complexity

- It is straight forward to identify linearly independent paths of simple programs. However, for complicated programs it is not easy to determine the number of independent paths.
- It provides a practical way of determining the number of linearly independent paths of a program.
- Given a control flow graph G,

Cyclomatic complexity V(G) is given as

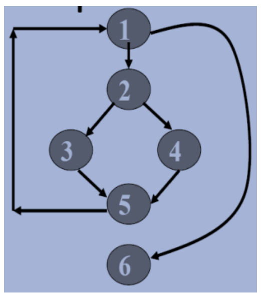
V(G)= E-N+2

Where N is the number of nodes in G

E is the number of edges in G

Example

Consider the following control-flow graph.



Cyclomatic complexity V(G) is

V(G) = 7-6+2

= 3

Another way of computing cyclomatic complexity is based on number of bounded areas in the graph.

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

For the above graph V(G) = 2+1 = 3

Another way of computing cyclomatic complexity is based on number of predicates (or decisions) in the graph.

V(G) = Number of predicates + 1

For the above graph V(G) = 2+1 = 3

6.5.2 Data Flow Testing

- Data flow testing uses the control flow graph to explore the unreasonable things that can happen to data (data flow anomalies).
- Data flow anomalies are detected based on the associations between values and variables. Some anomalies are
  - Variables are used without being initialized
  - Initialized variables are not used once

Data Object Categories

(d) Defined, Created, Initialized

(k) Killed, Undefined, Released

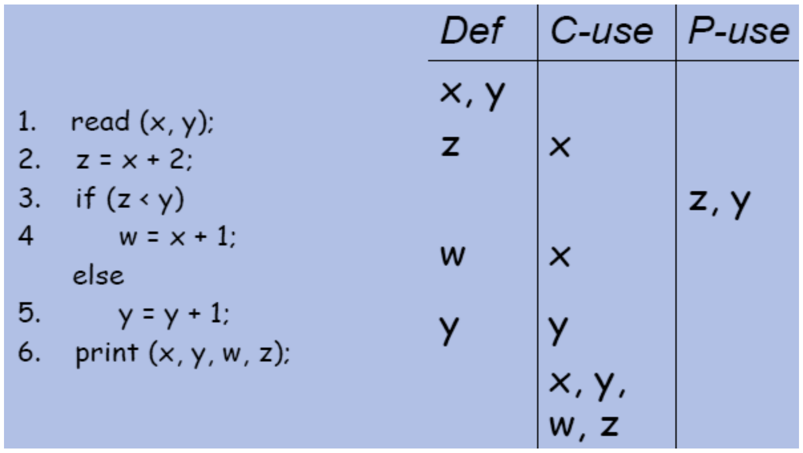
(u) Used:

– (c) Used in a calculation

– (p) Used in a predicate

Example

Consider the following sample program.



	Def	C-use	P-use
1. read (x, y);	x, y		
2. z = x + 2;	z	x	
3. if (z < y)			z, y
4 w = x + 1;	w	x	
else			
5. y = y + 1;	y	y	
6. print (x, y, w, z);		x, y, w, z	

6.5.3 Mutation Testing

- In this, software is first tested: Using an initial test suite designed using white box strategies we already discussed.
- After the initial testing is complete, mutation testing is taken up.
- The idea is insert faults into a program and then check whether the test suite is able to detect these.
- Each time the program is changed: It is called a mutated program and the change is called a **mutant.**
- Example primitive changes to a program are:

  –Deleting a statement

  –Altering an arithmetic operator,

  –Changing the value of a constant,

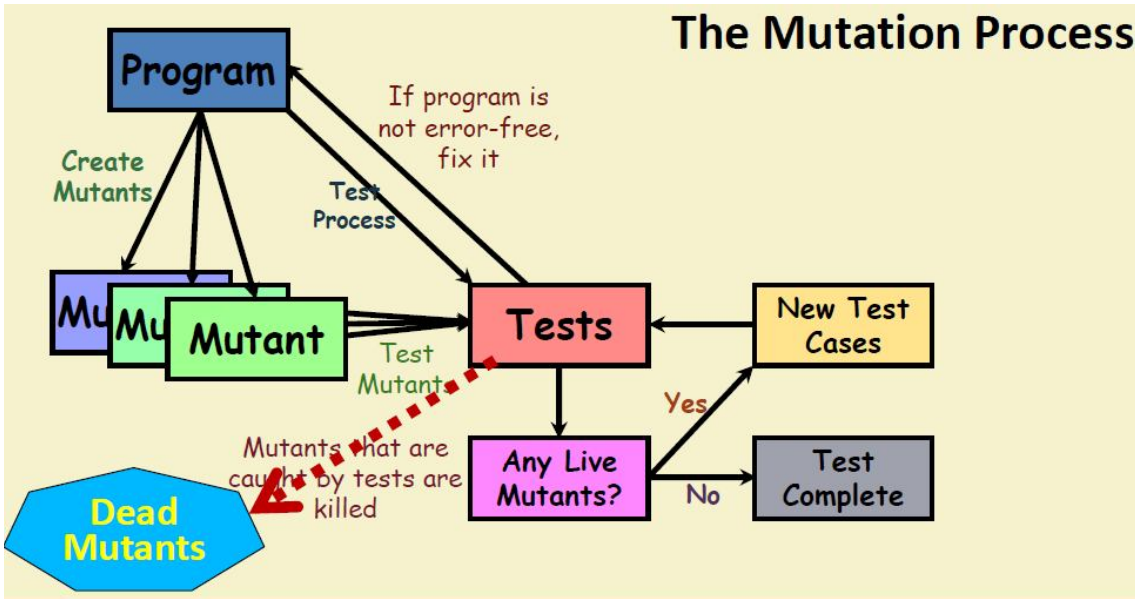
  –Changing a data type, etc.
- A mutated program is tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:

  –A mutant gives an incorrect result,

  –Then the mutant is said to be dead
- If a mutant remains alive even after all test cases have been exhausted,

  –The test suite is enhanced to kill the mutant.

- The mutation testing is shown below.



6.6. Metrics for Testing

- A few natural questions arise while testing:
  – How good is the testing that has been done?
  – What is the quality or reliability of software after testing is completed?

6.6.1 Coverage Analysis

- The coverage requirement at unit level can be 90% to 100%
- Besides the coverage of program constructs, coverage of requirements is also often examined.

6.6.2 Reliability

- As reliability of software depends considerably on the quality of testing, by assessing reliability we can also judge the quality of testing.
- Reliability estimation can be used to decide whether enough testing has been done.
- Let X be the random variable that represents the life of a system.
- Reliability of a system is the probability that the system has not failed by time t.
- In other words

$$R(t) = P(X > t).$$

6.6.3 Defect Removal Efficiency

- Defect Removal Efficiency is the ability of the system to remove defects prior to release.
- It is calculated as ratio of defects resolved to no of defects found.

*****

UNIT –VI

Assignment-Cum-Tutorial Questions

Objective Questions

1. White-box testing, sometimes called __________________.
2. The testing technique that requires preparing test cases to exercise the internal logic of a software module is [ ]
a) Behavioural Software Testing b) Black-box Testing
c) Grey-box Testing d) White-box Testing

3. White-box testing uses the ___________ structure of the procedural design to derive test cases. [ ]
a) Behaviour b) Control
c) Ariel d) None of the mentioned
4. Which one of the following testing techniques is effective in testing whether a developed software meets its non-functional requirements? [ ]
a) Path testing b) Dataflow testing
c) Robust boundary-value testing d) Performance testing
5. Which one of the following is a fault-based testing technique? [ ]
a) Pair wise testing b) Dataflow testing
c) Path testing d) Mutation testing
6. Suppose a certain function takes 5 Integer input parameters. The valid values for each parameter takes an integer value in the range 1..100. What is the minimum number of test cases required for robust boundary value testing? [ ]
a) 20 b) 21 c) 30 d) 31
7. Scenario coverage testing can be considered to be which one of the following types of testing strategies? [ ]
a) Pair-wise testing
b) Decision table-based testing
c) Equivalence partitioning-based testing
d) Boundary value-based testing
8. Which one of the following types of bugs may not get detected in black-box testing, but are very likely to be get detected by white-box testing? [ ]
a) Syntax errors b) Behavioral errors
c) Logic errors d) Performance errors
9. Cause-effect test cases are, in effect, designed using which one of the following types of testing techniques? [ ]
a) Decision-table based testing b) Coverage-based testing

c) Fault-based testing d) Path-based testing

10. If a user interface has three checkboxes, at least how many test cases are required to achieve pair-wise coverage? [ ]

a) 3 b) 4 c) 5 d) 6

11. Among the following test techniques, which one of the following is the strogest? [ ]

a) All path coverage testing b) Decision coverage testing
c) Basic condition coverage testing d) MC/DC testing

SECTION-B

Descriptive Questions

1) Explain different programming principles and guidelines on publicly available standards.
2) (a) Differentiate Error, Fault and Failure.
   (b) What is Test Case and Test Criteria?
3) Explain Cause-Effect Graph technique with decision table.
4) Explain about Mutation Testing and write the steps for mutation testing.
5) Analyze boundary value Analysis with formulas.
6) Apply state based testing for any example and draw state model and state table.
7) Identify def, C-use and P-use in data-flow based testing and draw data-flow graph for any example.
8) A program takes an angle as input within the range [0,360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.
9) What would be the Cyclomatic complexity of the following program?

```
int find-maximum(int i, int j, int k){
int max;
if(i>j) then
if(i>k) then max=I;
else max=k;
else if(j>k) max=j
else max=k;
return(max);
}
```

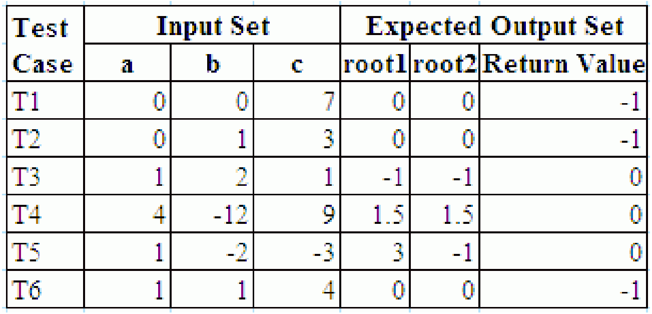
10) Sketch Reliability Model for failure intensity and also with respect to time.

C) Previous GATE Questions:

1) The following is the comment written for a C function.

```
/* This function computes the roots of a quadratic equation
        a.x^2 + b.x + c = . The function stores two real roots
        in *root1 and *root2 and returns the status of validity
```

of roots. It handles four different kinds of cases.
(i) When coefficient a is zero irrespective of discriminant
(ii) When discreminant is positive
(iii) When discriminant is zero
(iv) When discriminant is negative.
Only in case (ii) and (iii) the stored roots are valid.
Otherwise 0 is stored in roots. The function returns
0 when the roots are valid and -1 otherwise.
The function also ensures root1 >= root2
int get_QuadRoots(float a, float b, float c,
float *root1, float *root2);
*/
A software test engineer is assigned the job of doing black box testing. He comes up with the following test cases, many of which are redundant.



Test	Input Set			Expected Output Set		
Case	a	b	c	root1	root2	Return Value
T1	0	0	7	0	0	-1
T2	0	1	3	0	0	-1
T3	1	2	1	-1	-1	0
T4	4	-12	9	1.5	1.5	0
T5	1	-2	-3	3	-1	0
T6	1	1	4	0	0	-1

Which one of the following option provide the set of non-redundant tests using equivalence class partitioning approach from input perspective for black box testing?

A) T1, T2, T3, T6 **(GATE 2011)**

B) T1, T3, T4, T5

C) T2, T4, T5, T6

D) T2, T3, T4, T5

2) The following program is to be tested for statement coverage:

```
begin
  if (a== b) {S1; exit;}
  else if (c== d) {S2;]
       else {S3; exit;}
  S4;
end
```

The test cases T1, T2, T3 and T4 given below are expressed in terms of the properties satisfied by the values of variables a, b, c and d. The exact values are not given. T1 : a, b, c and d are all equal T2 : a, b, c and d are all distinct T3 : a = b and c != d T4 : a != b and c = d Which of the test suites given below ensures coverage of statements S1, S2, S3 and S4?

A) T1, T2, T3 (**GATE 2010)**

B) T2, T4

C) T3, T4

D) T1, T2, T4

3) Match the following:

List-I	List-II
a. Condition coverage	1. Black-box testing
b. Equivalence class partitioning	2. System testing
c. Volume testing	3. White-box testing
d. Alpha testing	4. Performance testing

A) a - 2 b - 3 c - 1 d - 4 (**GATE 2015)**

B) a - 3 b - 4 c - 2 d - 1

C) a - 3 b - 1 c - 4 d - 2

D) a - 3 b - 1 c - 2 d – 4