2018-19 Onwards (MR-18)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. IV Semester		
Code: 80511	SOFTWARE ENGINEERING	L	T	P
Credits: 3		3	-	-

Prerequisites: NIL

Course Objectives:

Student will be able to learn fundamental aspects of Software Engineering and analyze various process models. To identify various types of requirements and the process for Requirements Engineering. To make use of various System Models to conceptualize and construct a system. To demonstrate different testing tactics and define metrics for software measurement. To classify and mitigate the Software Risks and learn to achieve quality standards.

Module I: Introduction to Software Engineering [09 Periods]

Basics terms of Software Engineering: Evolving role of software, changing nature of Software, Software Myths. A Generic View of Process:-Software engineering-A layered technology, The Capability Maturity Model Integration (CMMI)

Process Models: The water fall model, Incremental process models, evolutionary process models, and the unified process.

Module II: Requirements of Software Engineering [09Periods]

Software Requirements: Functional and non functional requirements, User requirements, System requirements, Interface specification, The software requirements document.

Requirements Engineering Process: Feasibility studies, requirements elicitation and analysis, requirements validation, requirements management

Module III: Phases of Software Engineering [09 Periods]

A: System Models: Context models, Behavioral models, Data models, Object models, Structured methods

B: Design Engineering: Design process and design quality, design concepts the design model

Creating an architectural design: Software architecture, data design, architectural styles and patterns, architectural design

Module IV: Test Strategies [09 Periods]

Methods of Testing: A strategic approach to software testing, Black box and White box Testing, Validation Testing, System Testing.

Product Metrics: Software Quality, Metrics for analysis model, Metrics for design model, Metrics for source code, Metrics for testing, Metrics for maintenance

Metrics for process and products: Software measurement, Metrics for software quality

Module V: Risk Management [09Periods]

Management of Risk Process: Reactive Vs proactive risk strategies, Software risks, Risk identification, Risk projection Risk refinement, RMMM, RMMM plan

Quality Management: Quality concepts, Software quality assurance, Software reviews, Formal technical reviews, Statistical Software Quality Assurance, Software Reliability, ISO 9000 Quality standards

TEXT BOOKS:

1. Roger S. Pressman, Software engineering- A practitioner‘s Approach, McGraw Hill International Edition, 5th edition, 2001.
2. Ian Summerville, Software engineering, Pearson education Asia, 6th edition,2000.

REFERENCES:

1. Pankaj Jalote- An Integrated Approach to Software Engineering, Springer Verlag,1997.
2. James F Peters and WitoldPedryez, ―Software Engineering – An Engineering Approach, John Wiley and Sons, New Delhi,2000.
3. AliBehforooz and Frederick J Hudson, ―Software Engineering Fundamentals, Oxford University Press, New Delhi,1996.

E RESOURCES:

1. https://books.google.co.in/books?id=bL7QZHtWvaUC&printsec=frontcover&dq= software+engineering+by+roger+pressman+vth+edition+free+download&hl=en& sa=X&ved=0ahUKEwiLkOz-pL_TAhWIuI8KHZSxD2cQ6AEIMDAC#v=one page&q&f=false
2. https://books.google.co.in/books?id=PqsWaBkFh1wC&printsec=frontcover&dq= software+engineering+by+ian+sommerville+FREE+download&hl=en&sa=X&ve d=0ahUKEwjjv5fhpb_TAhUHOo8KHY5OAC4Q6AEIKjAB#v=onepage&q=soft ware%20engineering%20by%20ian%20sommerville%20FREE%20download&f= false
3. http://ieeexplore.ieee.org/document/4807670/
4. https://link.springer.com/search?facet-journal id=40411&package=open accessarticles&query=&facet-sub-discipline=%22Software+Engineering%22
5. http://freevideolectures.com/Course/2318/Software-Engineering
6. http://freevideolectures.com/Course/2318/Software-Engineering/5

Course Outcomes:

At the end of the course, students will be able to:

1. **Analyze** the customer business requirements and choose the appropriate Process model for the given project
2. **Elicit** functional and non-functional requirements using rigorous engineering methodology
3. **Conceptualize** and achieve requirements defined for the system using Architectural styles and Design patterns
4. **Design** test cases and define metrics for standardization.
5. **Assess**, mitigate and monitor the risks and assuring quality standards

LECTURE NOTES

ON

SOFTWARE ENGINEERING

II B. Tech II semester

MALLA REDDY ENGINEERING (A)

Module -I

INTRODUCTION TO SOFTWARE ENGINEERING

Basics terms of Software Engineering:-

Software: Software is

(1) Instructions (computer programs) that provide desired features, function, and performance, when executed
(2) Data structures that enable the programs to adequately manipulate information,
(3) Documents that describe the operation and use of the programs.

Characteristics of Software:

(1) Software is developed or engineered; it is not manufactured in the classical sense.
(2) Software does not "wear out"
(3) Although the industry is moving toward component-based construction, most software continues to be custom built.

Software Engineering:

(1) The systematic, disciplined quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
(2) The study of approaches as in (1)

EVOLVING ROLE OF SOFTWARE:-

Software takes dual role. It is both a **product** and a **vehicle** for delivering a product.

As a **product**: It delivers the computing potential embodied by computer Hardware or by a network of computers.

As a **vehicle**: Itis information transformer-producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as single bit or as complex as a multimedia presentation. Software delivers the most important product of our time-information.

- It transforms personal data
- It manages business information to enhance competitiveness
- It provides a gateway to worldwide information networks
- It provides the means for acquiring information

The role of computer software has undergone significant change over a span of little more than 50 years

- Dramatic Improvements in hardware performance
- Vast increases in memory and storage capacity
- A wide variety of exotic input and output options

1970s and 1980s:

- *Osborne*characterized a "new industrial revolution"
- *Toffler* called the advent of microelectronics part of "the third wave of change" in human history
- *Naisbitt* predicted the transformation from an industrial society to an "information society"
- *Feigenbaum and McCorduck* suggested that information and knowledge would be the focal point for power in the twenty-first century
- *Stoll* argued that the "electronic community" created by networks and software was the key to knowledge interchange throughout the world

1990s began:

- *Toffier* described a "power shift" in which old power structures disintegrate as computers and software lead to a "democratization of knowledge".
- *Yourdon* worried that U.S companies might lose their competitive edge in software related business and predicted "the decline and fall of the American programmer".
- *Hammer and Champy* argued that information technologies were to play a pivotal role in the "reengineering of the corporation".

Mid-1990s:

- The pervasiveness of computers and software spawned a rash of books by neo-luddites.

Later 1990s:

- *Yourdon* reevaluated the prospects of the software professional and suggested "the rise and resurrection" of the American programmer.
- The impact of the Y2K "time bomb" was at the end of 20th century

2000s progressed:

- *Johnson* discussed the power of "emergence" a phenomenon that explains what happens when interconnections among relatively simple entities result in a system that "self-organizes to form more intelligent, more adaptive behavior".
- *Yourdon* revisited the tragic events of 9/11 to discuss the continuing impact of global terrorism on the IT community
- *Wolfram* presented a treatise on a "new kind of science" that posits a unifying theory based primarily on sophisticated software simulations
- *Daconta* and his colleagues discussed the evolution of "the semantic web".

Today a huge software industry has become a dominant factor in the economies of the industrialized world.

THE CHANGING NATURE OF SOFTWARE:-

The 7 broad categories of computer software present continuing challenges for software engineers:

1) System software
2) Application software
3) Engineering/scientific software
4) Embedded software
5) Product-line software
6) Web-applications
7) Artificial intelligence software.

1. **System software:** System software is a collection of programs written to service other programs. The systems software is characterized by
   - heavy interaction with computer hardware
   - heavy usage by multiple users
   - concurrent operation that requires scheduling, resource sharing, and sophisticated process management
   - complex data structures
   - multiple external interfaces

   E.g. compilers, editors and file management utilities.

2. **Application software:**
   - Application software consists of standalone programs that solve a specific business need.
   - It facilitates business operations or management/technical decision making.
   - It is used to control business functions in real-time

   E.g. point-of-sale transaction processing, real-time manufacturing process control.

3. **Engineering/Scientific software:** Engineering and scientific applications range
   - from astronomy to volcanology
   - from automotive stress analysis to space shuttle orbital dynamics
   - from molecular biology to automated manufacturing

   computer aided design, system simulation and other interactive applications.

4. **Embedded software:**
   - Embedded software resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself.
   - It can perform limited and esoteric functions or provide significant function and control capability.

E.g. Digital functions in automobile, dashboard displays, braking systems etc.

5. **Product-line software:** Designed to provide a specific capability for use by many different customers, product-line software can focus on a limited and esoteric market place or address mass consumer markets

   E.g. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications

6. **Web-applications:** WebApps are evolving into sophisticated computing environments that not only provide standalone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

7. **Artificial intelligence software:** AI software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Application within this area includes robotics, expert systems, pattern recognition, artificial neural networks, theorem proving, and game playing.

The following are the **new challenges** on the horizon:

- **Ubiquitous computing**
- **Net sourcing**
- **Open source**
- **The "new economy"**

Ubiquitous computing: The **challenge** for software engineers will be to develop systems and application software that will allow small devices, personal computers and enterprise system to communicate across vast networks.

Net sourcing: The **challenge** for software engineers is to architect simple and sophisticated applications that provide benefit to targeted end-user market worldwide.

Open Source: The **challenge** for software engineers is to build source that is self descriptive but more importantly to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

The "new economy": The **challenge** for software engineers is to build applications that will facilitate mass communication and mass product distribution.

SOFTWARE MYTHS:-

Beliefs about software and the process used to build it- can be traced to the earliest days of computing myths have a number of attributes that have made them insidious.

1.Management myths:

Manages with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Myth: We already have a book that's full of standards and procedures for building software - Wont that provide my people with everything they need to know?

Reality: The book of standards may very well exist but, is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice?

Myth: If we get behind schedule, we can add more programmers and catch up.

Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who were working must spend time educating the new comers, thereby reducing the amount of time spend on productive development effort. People can be added but only in a planned and well coordinated manner.

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

2.Customer myths:

The customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations and ultimately, dissatisfaction with the developer.

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

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is recipe for disaster.

Myth: Project requirements continually 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 and change can cause upheaval that requires additional resources and major design modification.

3.Practitioner's myths:

Myths that are still believed by software practitioners: during the early days of software, programming was viewed as an art from old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our jobs are done.

Reality: Someone once said that the sooner you begin writing code, the longer it'll take you to get done. Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: The only deliverable work product for a successful project is the working program.
Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides guidance for software support.

Myth: software engineering will make us create voluminous and unnecessary documentation and will invariably slows 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.

A GENERIC VIEW OF PROCESS:-

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY:

- Software engineering is a fully layered technology.
- To develop a software, we need to go from one layer to another.
- All these layers are related to each other and each layer demands the fulfillment of the previous layer.

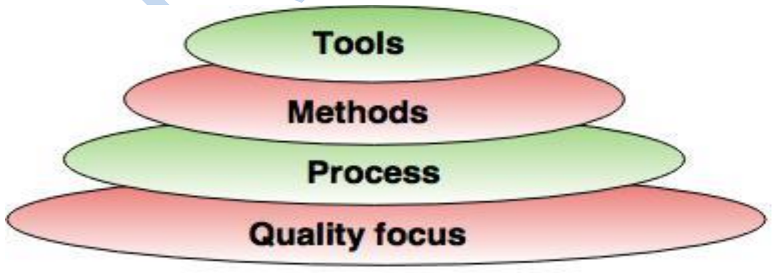


Fig. - Software Engineering Layers

The layered technology consists of:

1. **Quality focus**

The characteristics of good quality software are:

- Correctness of the functions required to be performed by the software.
- Maintainability of the software
- Integrity i.e. providing security so that the unauthorized user cannot access information or data.
- Usability i.e. the efforts required to use or operate the software.

2. **Process**

- It is the base layer or foundation layer for the software engineering.
- The software process is the key to keep all levels together.
- It defines a framework that includes different activities and tasks.
- In short, it covers all activities, actions and tasks required to be carried out for software development.

3. **Methods**

- The method provides the answers of all 'how-to' that are asked during the process.
- It provides the technical way to implement the software.
- It includes collection of tasks starting from communication, requirement analysis, analysis and design modelling, program construction, testing and support.

4. **Tools**

- The software engineering tool is an automated support for the software development.
- The tools are integrated i.e the information created by one tool can be used by the other tool.
- For example: The Microsoft publisher can be used as a web designing tool.

THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI):-

The CMMI represents a process meta-model in two different ways:

- As a continuous model
- As a staged model.

Each process area is formally assessed against specific goals and practices and is rated according to the following capability levels.

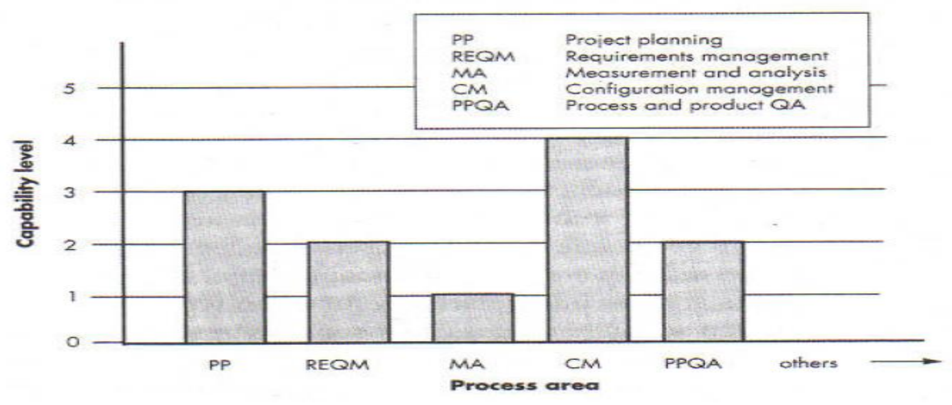


Figure , CMMI Process area capability profile

Level 0: Incomplete. The process area is either not performed or does not achieve all goals and objectives defined by CMMI for level 1 capability.

Level 1: Performed. All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed. All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed;

Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is "tailored from the organizations set of standard processes according to the organizations tailoring guidelines, and contributes and work products, measures and other process-improvement information to the organizational process assets".

Level 4: Quantitatively managed. All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment."Quantitative objectives for quality and process performance are established and used as criteria in managing the process"

Level 5: Optimized. All level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration"

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. Specific practices refine a goal into a set of process-related activities.

The specific goals (SG) and the associated specific practices(SP) defined for project planning are

SG 1 Establish estimates

SP 1.1 Estimate the scope of the project

SP 1.2 Establish estimates of work product and task attributes

SP 1.3 Define project life cycle

SP 1.4 Determine estimates of effort and cost

SG 2 Develop a Project Plan

SP 2.1 Establish the budget and schedule

SP 2.2 Identify project risks

SP 2.3 Plan for data management

SP 2.4 Plan for needed knowledge and skills

SP 2.5 Plan stakeholder involvement

SP 2.6 Establish the project plan

SG 3 Obtain commitment to the plan

SP 3.1 Review plans that affect the project

SP 3.2 Reconcile work and resource levels

SP 3.3 Obtain plan commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, **the generic goals (GG) and practices (GP)** for the project planning process area are

GG 1 Achieve specific goals

GP 1.1 Perform base practices

GG 2 Institutionalize a managed process

GP 2.1 Establish and organizational policy

GP 2.2 Plan the process

GP 2.3 Provide resources

GP 2.4 Assign responsibility

GP 2.5 Train people

GP 2.6 Manage configurations

GP 2.7 Identify and involve relevant stakeholders

GP 2.8 Monitor and control the process

GP 2.9 Objectively evaluate adherence

GP 2.10 Review status with higher level management

GG 3 Institutionalize a defined process

GP 3.1 Establish a defined process

GP 3.2 Collect improvement information

GG 4 Institutionalize a quantitatively managed process

GP 4.1 Establish quantitative objectives for the process

GP 4.2 Stabilize sub process performance

GG 5 Institutionalize and optimizing process

GP 5.1 Ensure continuous process improvement

GP 5.2 Correct root causes of problems

PROCESS MODELS

Prescriptive process models define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

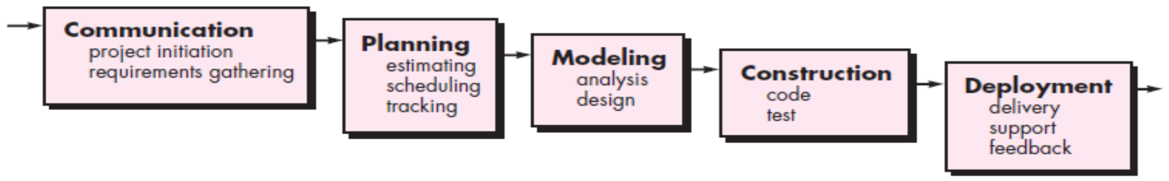
THE WATERFALL MODEL:-

The waterfall model, sometimes called the *classic life cycle*, suggests a systematic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.

Context: Used when requirements are reasonably well understood.

Advantage:

It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



Figure, Waterfall Model

The **problems** that are sometimes encountered when the waterfall model is applied are:

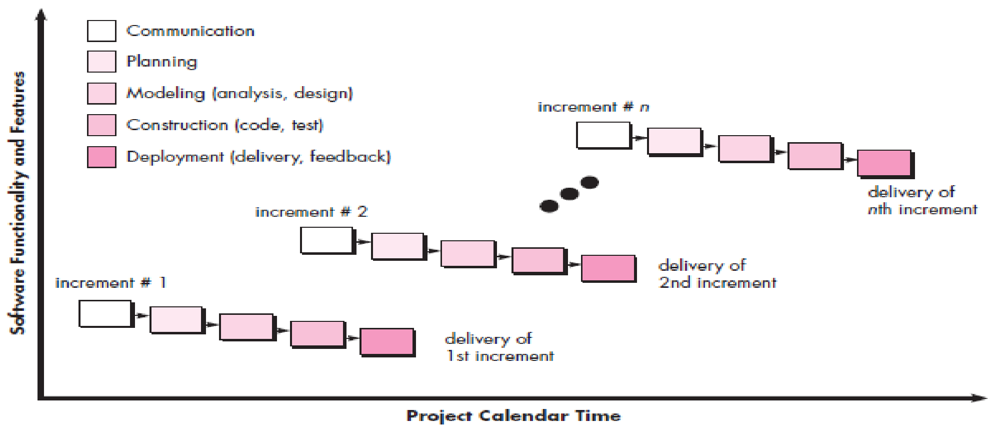
1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.
3. The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

INCREMENTAL PROCESS MODELS:-

1) The incremental model
2) The RAD model

THE INCREMENTAL MODEL:

Context: Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks.


Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
Software Functionality and Features
increment # 1
delivery of 1st increment
increment # 2
delivery of 2nd increment
increment # n
delivery of nth increment
Project Calendar Time

Figure, Incremental model

- The incremental model combines elements of the waterfall model applied in an iterative fashion.
- The incremental model delivers a series of releases called increments that provide progressively more functionality for the customer as each increment is delivered.
- When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed. The core product is used by the customer. As a result, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.

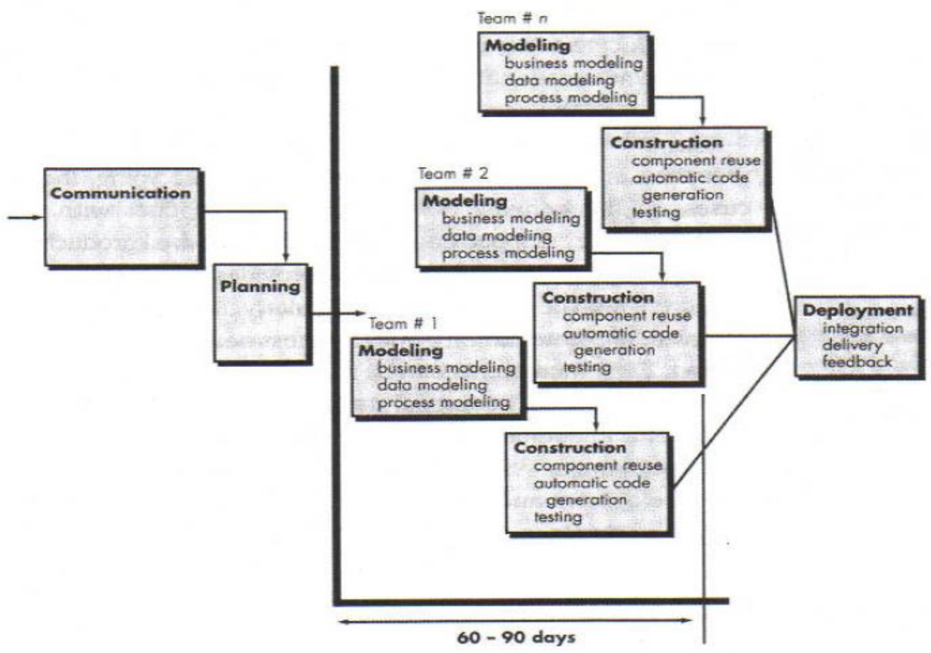
For *example*, word-processing software developed using the incremental paradigm might deliver basic file management editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

Difference: The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on delivery of an operational product with each increment

THE RAD MODEL:

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.

Context: If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within a very short time period.



Figure, The Rad model

The RAD approach maps into the generic framework activities.
Communication works to understand the business problem and the information characteristics that the software must accommodate.

Planning is essential because multiple software teams works in parallel on different system functions.

Modeling encompasses three major phases- business modeling, data modeling and process modeling- and establishes design representation that serve existing software components and the application of automatic code generation.

Deployment establishes a basis for subsequent iterations.

The RAD approach has **drawbacks**:

- For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail
- If a system cannot be properly modularized, building the components necessary for RAD will be problematic
- If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and
- RAD may not be appropriate when technical risks are high.

EVOLUTIONARY PROCESS MODELS:-

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

1.PROTOTYPING:

Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

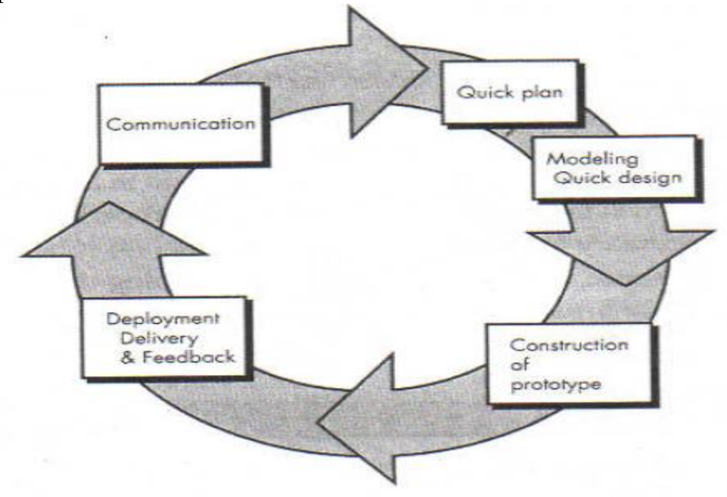


Figure , The prototyping model

Context:

If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.

If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

Advantages:

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.

Prototyping can be **problematic** for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development relents.
2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

2.THE SPIRAL MODEL

- The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.
- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

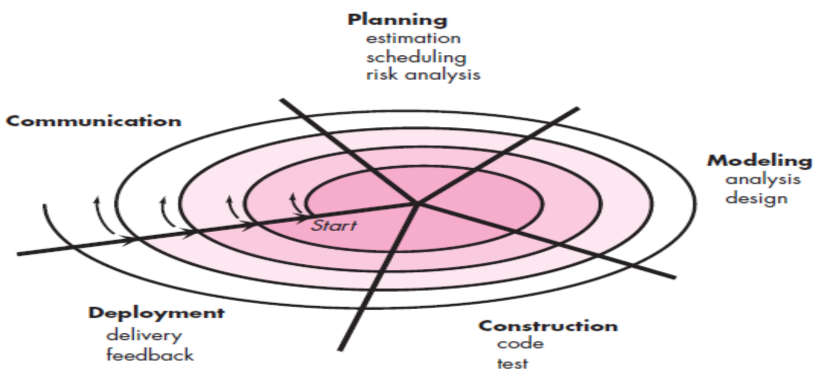


Figure , A typical spiral model

• **Anchor point milestones-** a combination of work products and conditions that are attained along the path of the spiral- are noted for each evolutionary pass.

- The first circuit around the spiral might result in the development of product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.

➢ The first circuit around the spiral might represent a "**concept development project**" which starts at the core of the spiral and continues for multiple iterations until concept development is complete.

➢ If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "**new product development project**" commences.

➢ Later, a circuit around the spiral might be used to represent a "**product enhancement project**." In essence, the spiral, when characterized in this way, remains operative until the software is retired.

Context: The spiral model can be adopted to apply throughout the entire life cycle of an application, from concept development to maintenance.

Advantages:

It provides the potential for rapid development of increasingly more complete versions of the software.

The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model uses prototyping as a risk reduction mechanism but, more importantly enables the developer to apply the prototyping approach at any stage in the evolution of the product.

Draw Backs:
The spiral model is not a panacea. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

3.THE CONCURRENT DEVELOPMENT MODEL:

The concurrent development model, sometimes called *concurrent engineering*, can be represented schematically as a series of framework activities, software engineering actions and tasks, and their associated states.

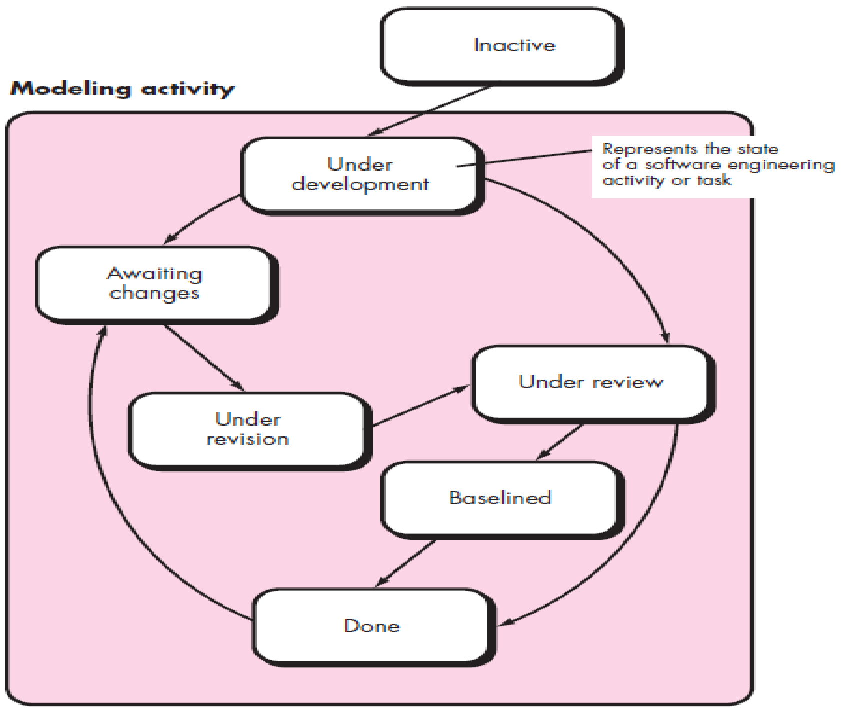


Figure , One element of the concurrent process model

The activity *modeling* may be in anyone of the states noted at any given time. Similarly, other activities or tasks can be represented in an analogous manner. All activities exist concurrently but reside in different states.

Any of the activities of a project may be in a particular state at any one time:

- under development
- awaiting changes
- under revision
- under review

In a project the *communication* activity has completed its first iteration and exists in the **awaiting changes** state. The modeling activity which existed in the **none** state while initial communication was completed, now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.

The event analysis model correction which will trigger the analysis action from the **done** state into the **awaiting changes** state.

Context: The concurrent model is often more appropriate for system engineering projects where different engineering teams are involved.

Advantages:

- The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.
- It defines a network of activities rather than each activity, action, or task on the network exists simultaneously with other activities, action and tasks.

A FINAL COMMENT ON EVOLUTIONARY PROCESSES:

- The concerns of evolutionary software processes are:
- The first concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.
- Second, evolutionary software process do not establish the maximum speed of the evolution. If the evolution occurs too fast, without a period of relaxation, it is certain that the process will fall into chaos.
- Third, software processes should be focused on flexibility and extensibility rather than on high quality.

THE UNIFIED PROCESS:-

The unified process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development.
The Unified process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse". If suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

i). A BRIEF HISTORY:

During the 1980s and into early 1990s, object-oriented (OO) methods and programming languages gained a widespread audience throughout the software engineering community. A wide variety of object-oriented analysis (OOA) and design (OOD) methods were proposed during the same time period.

During the early 1990s James Rumbaugh, Grady Booch, and Ival Jacobsom began working on a "Unified method" that would combine the best features of each of OOD & OOA. The result was UML- a unified modeling language that contains a robust notation fot the modeling and development of OO systems.

By 1997, UML became an industry standard for object-oriented software development. At the same time, the Rational Corporation and other vendors developed automated tools to support UML methods.

Over the next few years, Jacobson, Rumbugh, and Booch developed the Unified process, a framework for object-oriented software engineering using UML. Today, the Unified process and UML are widely used on OO projects of all kinds. The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

ii). PHASES OF THE UNIFIED PROCESS:

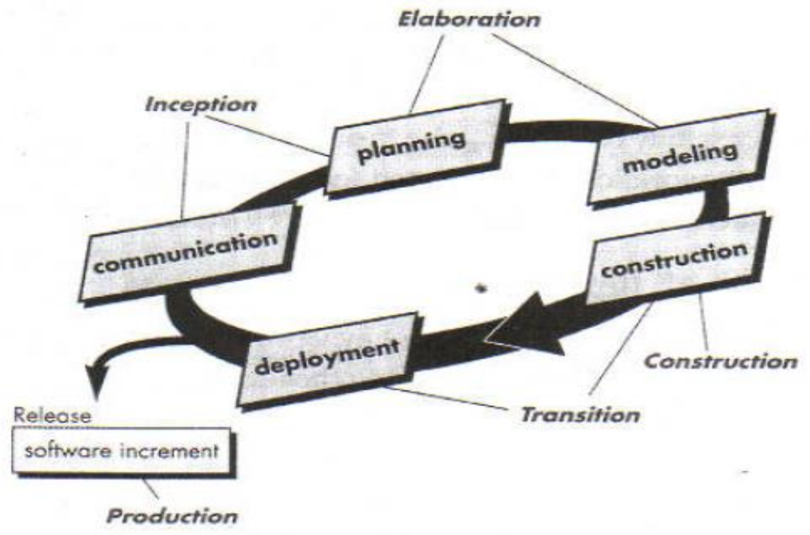
The ***inception*** phase of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end-users, business requirements for the software are identified, a rough architecture for the system is proposed and a plan for the iterative, incremental nature of the ensuing project is developed.

The ***elaboration*** phase encompasses the customer communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use-cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software- the use-case model, the analysis model, the design model, the implementation model, and the deployment model.

The ***construction*** phase of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use-case operational for end-users. To accomplish this, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment.

The ***transition*** phase of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment activity. Software given to end-users for beta testing, and user feedback reports both defects and necessary changes.

The ***production*** phase of the UP coincides with the deployment activity of the generic process. During this phase, the on-going use of the software is monitored, support for the operating environment is provided, and defect reports and requests for changes are submitted and evaluated.



Figure, The unified process

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set. That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.

iii). UNIFIED PROCESS WORK PRODUCTS:

During the ***inception phase***, the intent is to establish an overall "vision" for the project,

- identify a set of business requirements,
- make a business case for the software, and
- define project and business risks that may represent a threat to success.

The most important work product produced during the inception is the use-case modell-a collection of use-cases that describe how outside actors interact with the system and gain value from it. The use-case model is a collection of software features and functions by describing a set of preconditions, a flow of events and a set of post-conditions for the interaction that is depicted.

The use-case model is refined and elaborated as each UP phase is conducted and serves as an important input for the creation of subsequent work products. During the inception phase only 10 to 20 percent of the use-case model is completed. After elaboration, between 80 to 90 percent of the model has been created.

The ***elaboration phase*** produces a set of work products that elaborate requirements and produce and architectural description and a preliminary design. The UP analysis model is the work product that is developed as a consequence of this activity. The classes and analysis packages defined as part of the analysis model are refined further into a design model which identifies design classes, subsystems, and the interfaces between subsystems. Both the analysis and design models expand and refine an evolving representation of software architecture. In addition the elaboration phase revisits risks and the project plan to ensure that each remains valid.

The ***construction phase*** produces an implementation model that translates design classes into software components into the physical computing environment. Finally, a test model describes tests that are used to ensure that use cases are properly reflected in the software that has been constructed.

The ***transition phase*** delivers the software increment and assesses work products that are produced as end-users work with the software. Feedback from beta testing and qualitative requests for change is produced at this time.

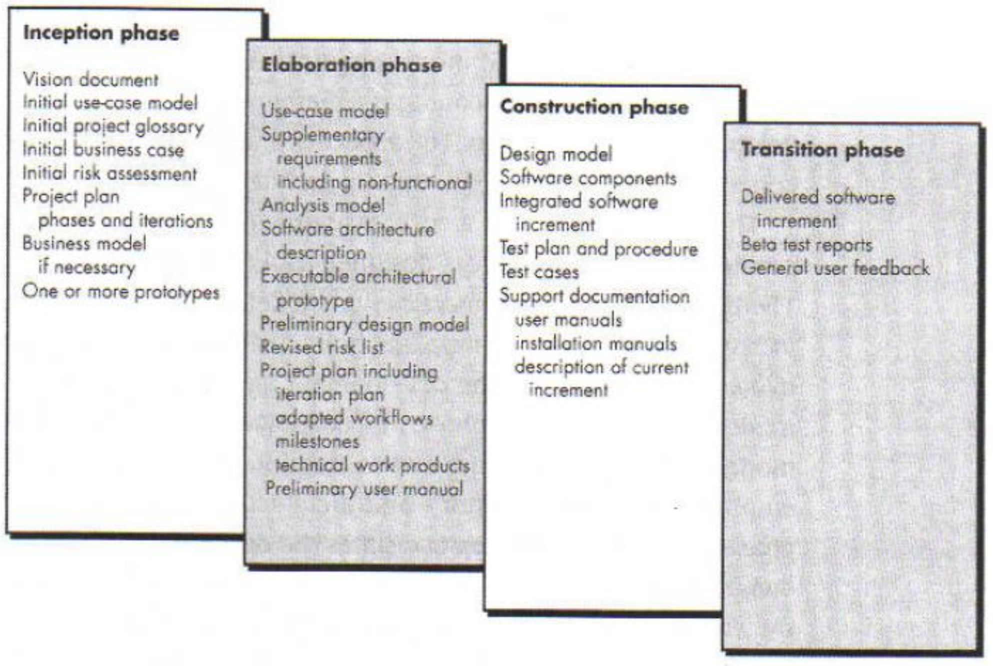


Figure , major work products produced for each UP phase

Module – II

Requirements of Software Engineering

SOFTWARE REQUIREMENTS:-

Software requirements are necessary

- To introduce the concepts of user and system requirements
- To describe functional and non-functional requirements
- To explain how software requirements may be organized in a requirements document

What is a requirement?

- The requirements for the system are the description of the services provided by the system and its operational constraints
- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
    - May be the basis for a bid for a contract - therefore must be open to interpretation;
    - May be the basis for the contract itself - therefore must be defined in detail;

Both these statements may be called requirements

Requirements engineering:

- The process of finding out, analyzing documenting and checking these services and constraints is called requirement engineering.
- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions ofthe system services and constraints that are generated during the requirements engineering process.

Requirements abstraction (Davis):

If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The **requirements** must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a **system definition** for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the **requirements document** for the system."

Types of requirement:

- **User requirements**
    - Statements in natural language plus diagrams ofthe services the system provides and its operational constraints. Written for customers.
- **System requirements**
    - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Definitions and specifications:

User Requirement Definition:

The software must provide the means of representing and accessing external files created by other tools.

System Requirement specification:

- The user should be provided with facilities to define the type of external files.
- Each external file type may have an associated tool which may be applied to the file.
- Each external file type may be represented as a specific icon on the user's display.
- Facilities should be provided for the icon representing an external file type to be defined by the user.
- When an user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

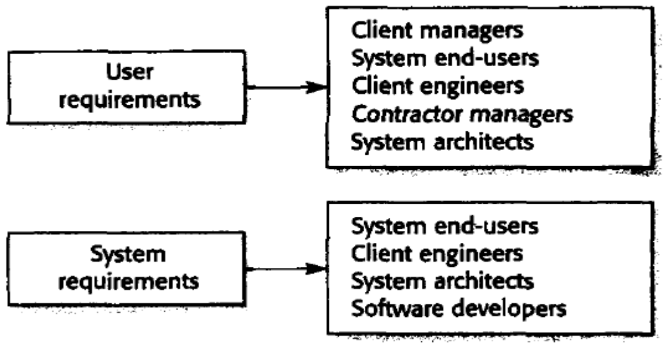


Figure ,readers of different types of Specification

Functional and non-functional requirements:

Functional requirements

- Statements of services the system should provide how the system should react to particular inputs and how the system should behave in particular situations.

Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

Domain requirements

- Requirements that come from the application domain of the system and that reflect characteristics of that domain.

FUNCTIONAL REQUIREMENTS:

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

The functional requirements for **The LIBSYS system:**

- A library system that provides a single interface to a number of databases of articles in different libraries.
- Users can search for, download and print these articles for personal study.

Examples of functional requirements

- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.
- Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

Requirements imprecision

- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.

- Consider the term ‘appropriate viewers’
    - User intention - special purpose viewer for each different document type;
    - Developer interpretation - Provide a text viewer that shows the contents of the document.

Requirements completeness and consistency:

In principle, requirements should be both complete and consistent.

Complete

- They should include descriptions of all facilities required.

Consistent

- There should be no conflicts or contradictions in the descriptions of the system facilities.

In practice, it is impossible to produce a complete and consistent requirements document.

NON-FUNCTIONAL REQUIREMENTS

- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

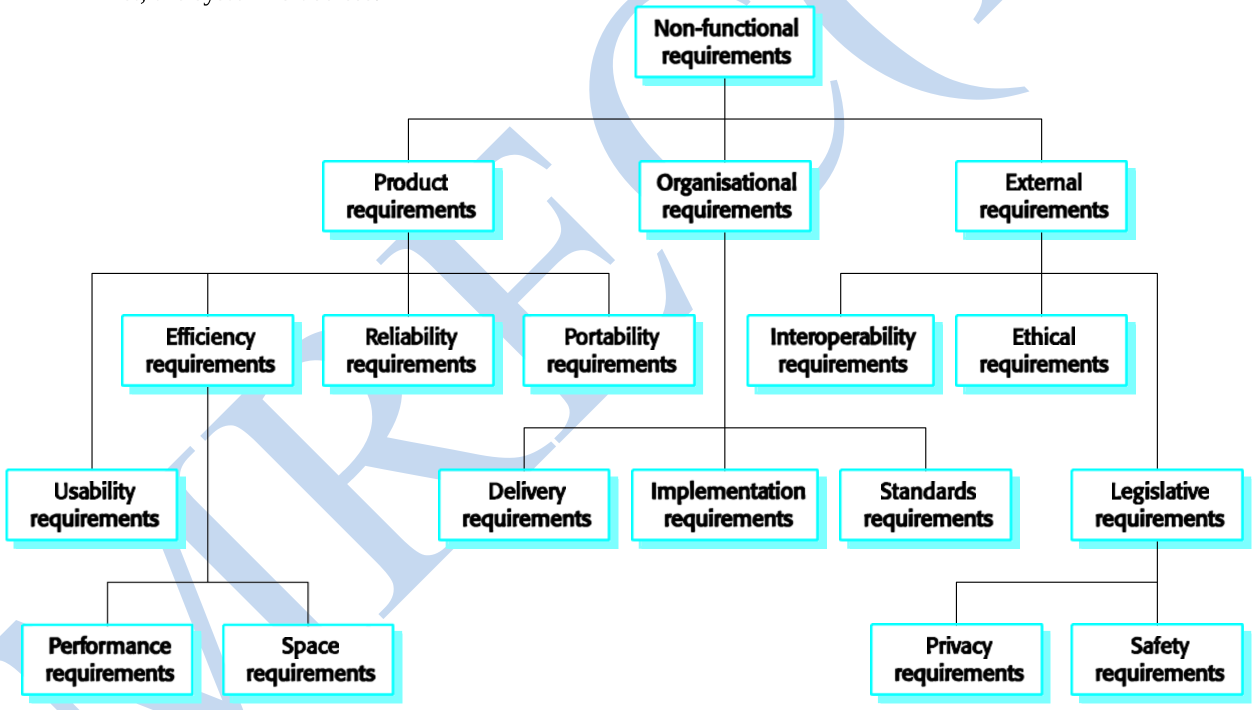


Figure , types of Non-functional requirement

The types of non-functional requirements are:

Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- *Eg:*The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.

Organisational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- *Eg:* The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

External requirements

- Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.
- *Eg:* The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Goals and requirements:

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
  - A general intention of the user such as ease of use.
  - The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.
- Verifiable non-functional requirement
  - A statement using some measure that can be objectively tested.
  - Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.
- Goals are helpful to developers as they convey the intentions of the system users.

Requirements measures:

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	M Bytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Requirements interaction:

- Conflicts between different non-functional requirements are common in complex systems.
- Spacecraft system
  - To minimise weight, the number of separate chips in the system should be minimised.
  - To minimise power consumption, lower power chips should be used.
- However, using low power chips may mean that more chips have to be used. Which is the most critical requirement?

A common **problem with non-functional requirements** is that they can be difficult to verify. Users or customers often state these requirements as general goals such as ease of use, the ability of the system to recover from failure or rapid user response. These vague goals cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered.

DOMAIN REQUIREMENTS

- Derived from the application domain and describe system characteristics and features that reflect the domain.
- Domain requirements be new functional requirements, constraints on existing requirements or define specific computations.
- If domain requirements are not satisfied, the system may be unworkable.

Library system domain requirements:

- There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.
- Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Domain requirements problems

Understandability

- Requirements are expressed in the language of the application domain;
- This is often not understood by software engineers developing the system.

Implicitness

- Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

USER REQUIREMENTS:-

- Should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.
- User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

Problems with natural language

Lack of clarity

- Precision is difficult without making the document difficult to read.

Requirements confusion

- Functional and non-functional requirements tend to be mixed-up.

Requirements amalgamation

- Several different requirements may be expressed together.

Requirement problems

Database requirements includes both conceptual and detailed information

- Describes the concept of a financial accounting system that is to be included in LIBSYS;

- However, it also includes the detail that managers can configure this system - this is unnecessary at this level.

Grid requirement mixes three different kinds of requirement

- Conceptual functional requirement (the need for a grid);
- Non-functional requirement (grid units);
- Non-functional UI requirement (grid switching).
- Structured presentation

Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.

SYSTEM REQUIREMENTS:-

- More detailed specifications of system functions, services and constraints than user requirements.
- They are intended to be a basis for designing the system.
- They may be incorporated into the system contract.
- System requirements may be defined or illustrated using system models

Requirements and design

In principle, requirements should state what the system should do and the design should describe how it does this.

In practice, requirements and design are inseparable

- A system architecture may be designed to structure the requirements;
- The system may inter-operate with other systems that generate design requirements;
- The use of a specific design may be a domain requirement.

Problems with NL(natural language) specification

Ambiguity

- The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult.

Over-flexibility

- The same thing may be said in a number of different ways in the specification.

Lack of modularisation

- NL structures are inadequate to structure system requirements.

Alternatives to NL specification:

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.

Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT. Now, use-case descriptions and sequence diagrams are commonly used .
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Structured language specifications:

- The freedom of the requirements writer is limited by a predefined template for requirements.
- All requirements are written in a standard way.
- The terminology used in the description may be limited.
- The advantage is that the most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.

Form-based specifications

- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Indication of other entities required.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

Tabular specification

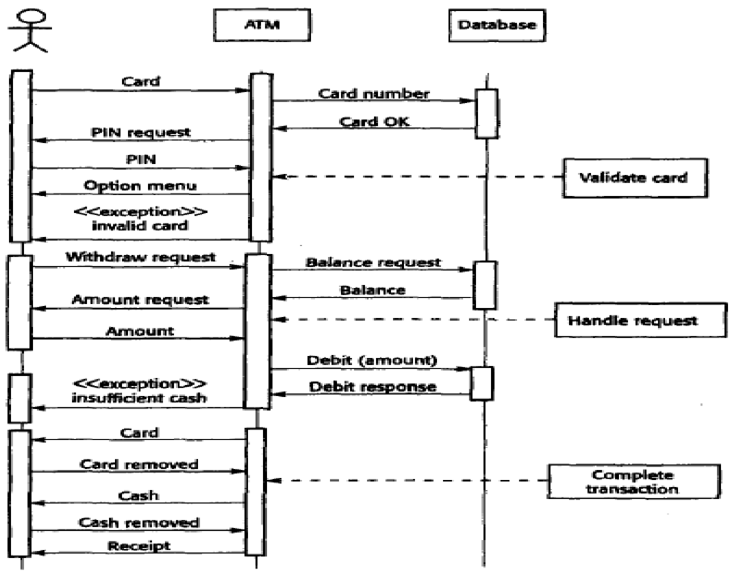
- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.

Graphical models

- Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.

Sequence diagrams

- These show the sequence of events that take place during some user interaction with a system.
- You read them from top to bottom to see the order of the actions that take place.
- Cash withdrawal from an ATM
  - Validate card;
  - Handle request;
  - Complete transaction.



Figure, Sequence diagram of ATM withdrawal

System requirement specification using a standard form:

1. Function
2. Description
3. Inputs
4. Source
5. Outputs
6. Destination
7. Action
8. Requires
9. Pre-condition
10. Post-condition
11. Side-effects

When a standard form is used for specifying functional requirements, the following information should be included:

1. Description of the function or entity being specified
2. Description of its inputs and where these come from
3. Description of its outputs and where these go to
4. Indication of what other entities are used
5. Description of the action to be taken
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called
7. Description of the side effects of the operation.

INTERFACE SPECIFICATION:-

- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
- Three types of interface may have to be defined
  - ***Procedural interfaces*** where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Applicatin Programming Interfaces (APIs)
  - ***Data structures that are exchanged*** that are passed from one sub-system to another. Graphical data models are the best notations for this type of description
  - ***Data representations*** that have been established for an existing sub-system
- Formal notations are an effective technique for interface specification.

THE SOFTWARE REQUIREMENTS DOCUMENT:-

- The requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

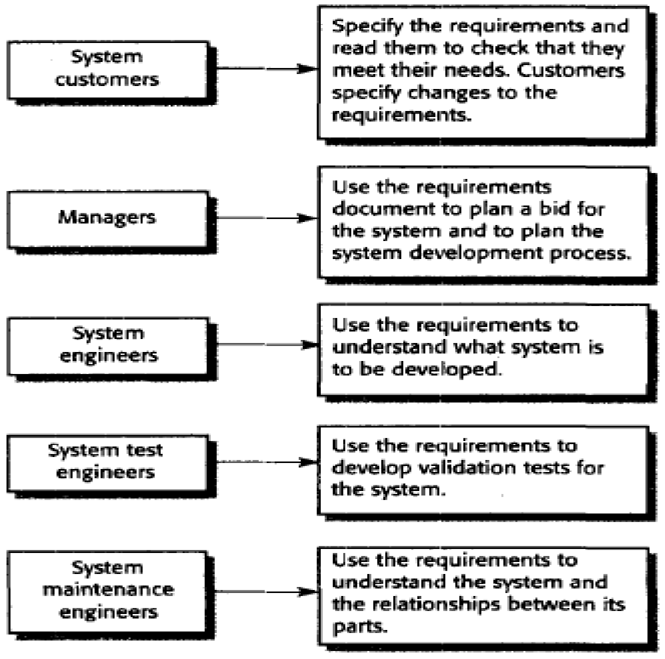
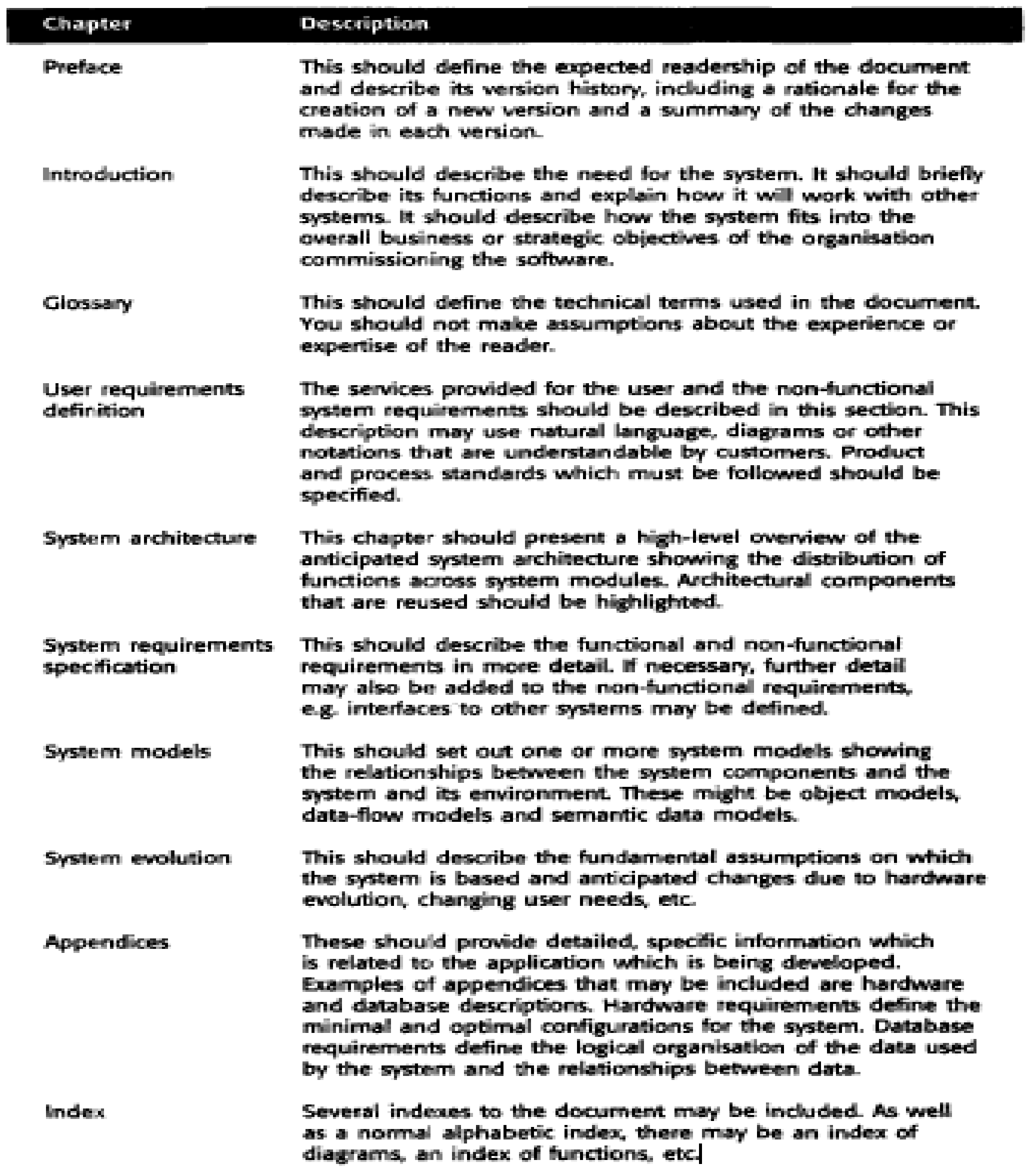

System customers
Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers
Use the requirements document to plan a bid for the system and to plan the system development process.
System engineers
Use the requirements to understand what system is to be developed.
System test engineers
Use the requirements to develop validation tests for the system.
System maintenance engineers
Use the requirements to understand the system and the relationships between its parts.

Figure , Users of a requirements document

IEEE requirements standard defines a generic structure for a requirements document that must be instantiated for each specific system.

1. Introduction.
   i) Purpose of the requirements document
   ii) Scope of the project
   iii) Definitions, acronyms and abbreviations
   iv) References
   v) Overview of the remainder of the document
2. General description.
   i) Product perspective
   ii) Product functions
   iii) User characteristics
   iv) General constraints
   v) Assumptions and dependencies
3. Specific requirements cover functional, non-functional and interface requirements. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.
4. Appendices.
5. Index.



Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe its functions and explain how it will work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organisation commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	The services provided for the user and the non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product and process standards which must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements, e.g. interfaces to other systems may be defined.
System models	This should set out one or more system models showing the relationships between the system components and the system and its environment. These might be object models, data-flow models and semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	These should provide detailed, specific information which is related to the application which is being developed. Examples of appendices that may be included are hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organisation of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.

Figure , The structure of a requirements documentation

Although the IEEE standard is not ideal, it contains a great deal of good advice on how to write requirements and how to avoid problems. It is too general to be an organisational standard in its own right. It is a general framework that can be tailored and adapted to define a standard geared to the needs of a particular organisation. **Figure** , illustrates a possible organisation for a requirements document that is based on the IEEE standard. However, I have extended this to include information about predicted system evolution. This was first proposed by Heninger (Heninger, 1980) and, as I have discussed, helps the maintainers of the system and may allow designers to include support for future system features

By contrast, when the software is part of a large system engineering project that includes interacting hardware and software systems, it is often essential to define the requirements to a fine level of detail. This means that the requirements documents are likely to be very long and should include most if not all of the chapters shown in **Figure** . For long documents, it is particularly important to include a comprehensive table of contents and document index so that readers can find the information that they need.

REQUIREMENTS ENGINEERING PROCESSES:-

The **goal** of requirements engineering process is to create and maintain a system requirements document. The overall process includes four high-level requirement engineering sub-processes. These are concerned with

- ✓ Assessing whether the system is useful to the business(feasibility study)
- ✓ Discovering requirements(elicitation and analysis)
- ✓ Converting these requirements into some standard form(specification)
- ✓ Checking that the requirements actually define the system that the customer wants(validation) The process of managing the changes in the requirements is called **requirement management**.

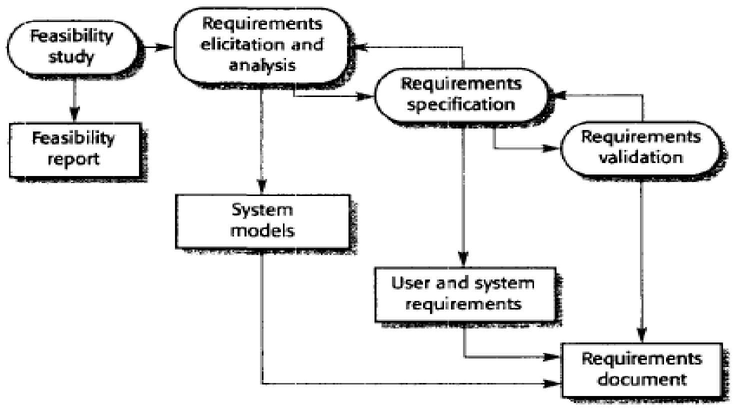
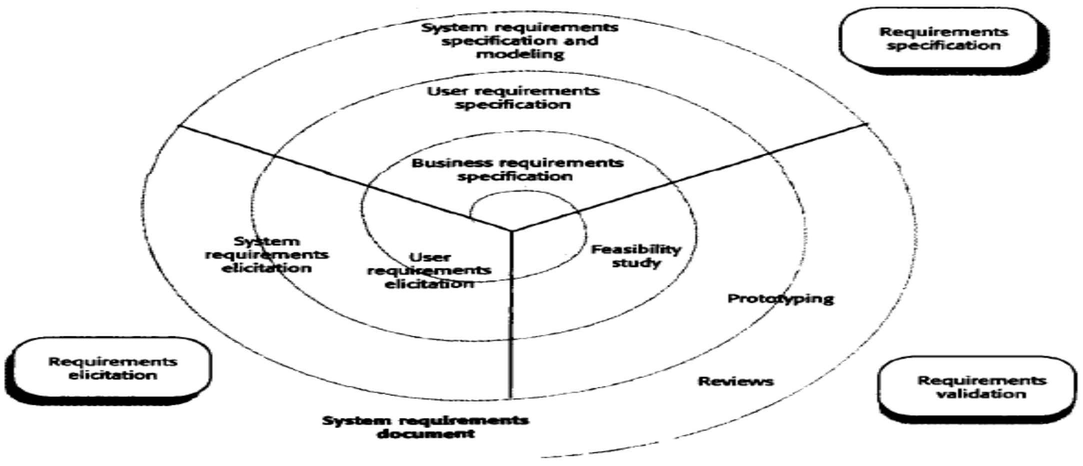


Figure , The requirements engineering process

Requirements engineering:

The alternative perspective on the requirements engineering process presents the process as a **three-stage activity** where the activities are organized as an iterative process around a spiral. The amount of time and effort devoted to each activity in iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements and the user requirements. Later in the process, in the outer rings of the spiral, more effort will be devoted to system requirements engineering and system modeling.



System requirements specification and modeling
Requirements specification
User requirements specification
Business requirements specification
System requirements elicitation
User requirements elicitation
Feasibility study
Prototyping
Requirements elicitation
Reviews
Requirements validation
System requirements document

Figure , Spiral model of requirements engineering processes

This spiral model accommodates approaches to development in which the requirements are developed to different levels of detail. The number of iterations around the spiral can vary, so the spiral can be exited after some or all of the user requirements have been elicited.

Some people consider requirements engineering to be the process of applying a structured analysis method such as object-oriented analysis. This involves analyzing the system and developing a set of graphical system models, such as use-case models, that then serve as a system specification. The set of models describes the behavior of the system and are annotated with additional information describing, for example, its required performance or reliability.

FEASIBILITY STUDIES:-

A feasibility study decides whether or not the proposed system is worthwhile. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it worth carrying on with the requirements engineering and system development process.

- A short focused study that checks
  - If the system contributes to organisational objectives;
  - If the system can be engineered using current technology and within budget;
  - If the system can be integrated with other systems that are used.

Feasibility study implementation:

- A feasibility study involves information assessment, information collection and report writing.
- Questions for people in the organisation
  - What if the system wasn’t implemented?
  - What are current process problems?
  - How will the proposed system help?
  - What will be the integration problems?
  - Is new technology needed? What skills?
  - What facilities must be supported by the proposed system?

In a feasibility study, you may consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system. They should try to complete a feasibility study in two or three weeks.

Once you have the information, you write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

REQUIREMENT ELICITATION AND ANALYSIS:-

The requirement engineering process is requirements elicitation and analysis.

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system’s operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Problems of requirements analysis

- Stakeholders don’t know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders mayemerge and the business environment change.

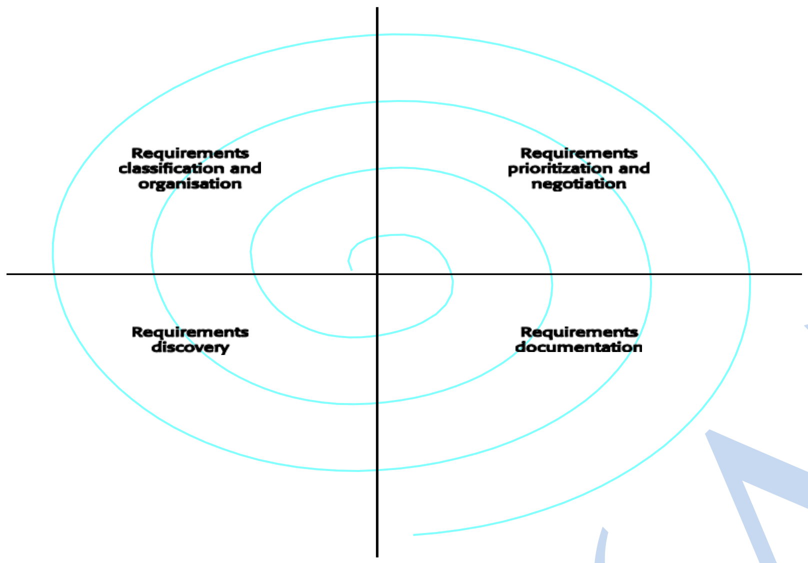

Requirements classification and organisation
Requirements prioritization and negotiation
Requirements discovery
Requirements documentation

Figure , The requirements elicitation and analysis process

Process activities

1. Requirements discovery
   - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
2. Requirements classification and organisation
   - Groups related requirements and organises them into coherent clusters.
3. Prioritisation and negotiation
   - Prioritising requirements and resolving requirements conflicts.
4. Requirements documentation
   - Requirements are documented and input into the next round of the spiral.

The process cycle starts with requirements discovery and ends with requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle.
Requirements classification and organization is primarily concerned with identifying overlapping requirements from different stakeholders and grouping related requirements. The most common way of grouping requirements is to use a model of the system architecture to identify subsystems and to associate requirements with each sub-system.
Inevitably, stakeholders have different views on the importance and priority of requirements, and sometimes these view conflict. During the process, you should organize regular stakeholder negotiations so that compromises can be reached.
In the requirement documenting stage, the requirements that have been elicited are documented in such a way that they can be used to help with further requirements discovery.

1. REQUIREMENTS DISCOVERY:

- Requirement discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems.
- They interact with stakeholders through interview and observation and may use scenarios and prototypes to help with the requirements discovery.
- Stakeholders range from system end-users through managers and external stakeholders such as regulators who certify the acceptability of the system.
- For example, system stakeholder for a bank ATM include
  1. Bank customers
  2. Representatives of other banks
  3. Bank managers
  4. Counter staff
  5. Database administrators
  6. Security managers

7. Marketing department
8. Hardware and software maintenance engineers
9. Banking regulators

Requirements sources(stakeholders, domain, systems) can all be represented as system viewpoints, where each viewpoints, where each viewpoint presents a sub-set of the requirements for the system.

Viewpoints:

- Viewpoints are a way of structuring the requirements to represent the perspectives of different stakeholders. Stakeholders may be classified under different viewpoints.
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements.

Types of viewpoint:

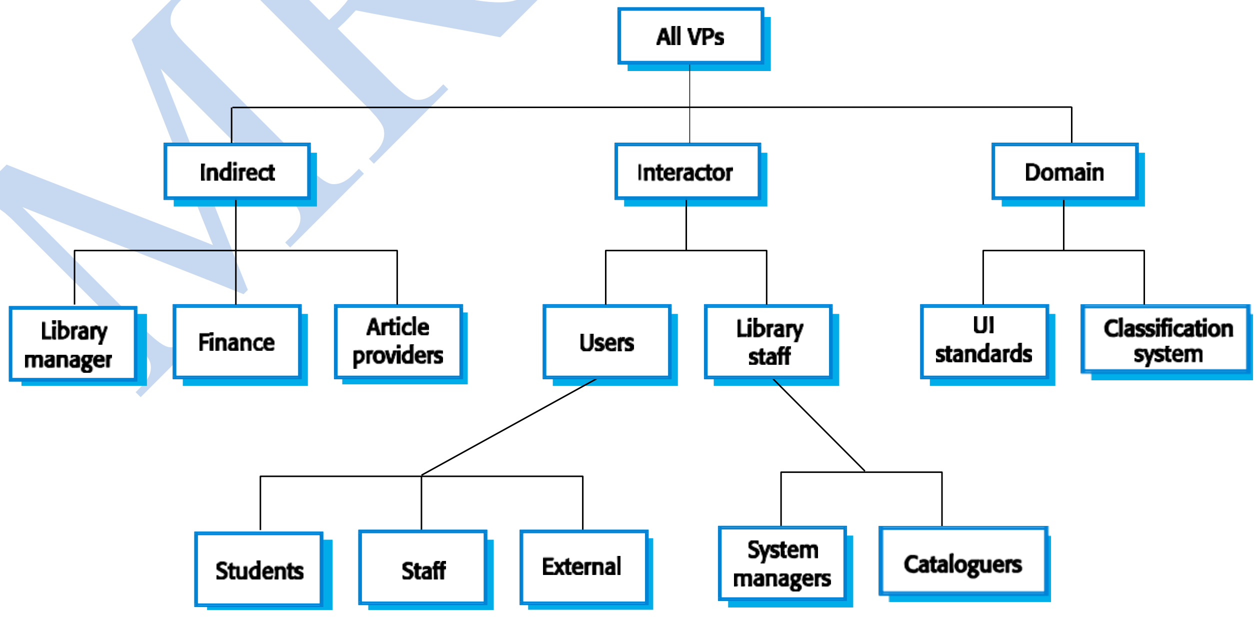
1. **Interactor viewpoints**
   – People or other systems that interact directly with the system. These viewpoints provide detailed system requirements covering the system features and interfaces. In an ATM, the customer's and the account database are interactor VPs.
2. **Indirect viewpoints**
   – Stakeholders who do not use the system themselves but who influence the requirements. These viewpoints are more likely to provide higher-level organisation requirements and constraints. In an ATM, management and security staff are indirect viewpoints.
3. **Domain viewpoints**
   – Domain characteristics and constraints that influence the requirements. These viewpoints normally provide domain constraints that apply to the system. In an ATM, an example would be standards for inter-bank communications.

Typically, these viewpoints provide different types of requirements.

Viewpoint identification:

- Identify viewpoints using
  – Providers and receivers of system services;
  – Systems that interact directly with the system being specified;
  – Regulations and standards;
  – Sources of business and non-functional requirements.
  – Engineers who have to develop and maintain the system;
  – Marketing and other business viewpoints.

LIBSYS viewpoint hierarchy



Interviewing

In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.

There are two types of interview

- **Closed interviews** where a pre-defined set of questions are answered.
- **Open interviews** where there is no pre-defined agenda and a range ofissues are explored with stakeholders.
-

Interviews in practice:

- Normally a mix of closed and open-ended interviewing.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Effective interviewers:

- Interviewers should be open-minded, willing to listen to stakeholders and should not have pre-conceived ideas about the requirements.
- They should prompt the interviewee with a question or a proposal and should not simplyexpect them to respond to a question such as 'what do you want'.
-

Scenarios:

Scenarios are real-life examples of how a system can be used.

- They should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.

Use cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

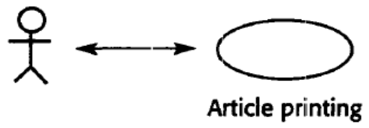


Figure A , A simple use-case for article printing

Figure A, illustrates the essentials of the use-case notation. Actors in the process are represented as stick figures, and each class of interaction is represented as a named ellipse. The set of use-cases represents all of the possible interactions to be represented in the system requirements. Figure B, develops the LIBSYS example and shows other use-cases in that environment.

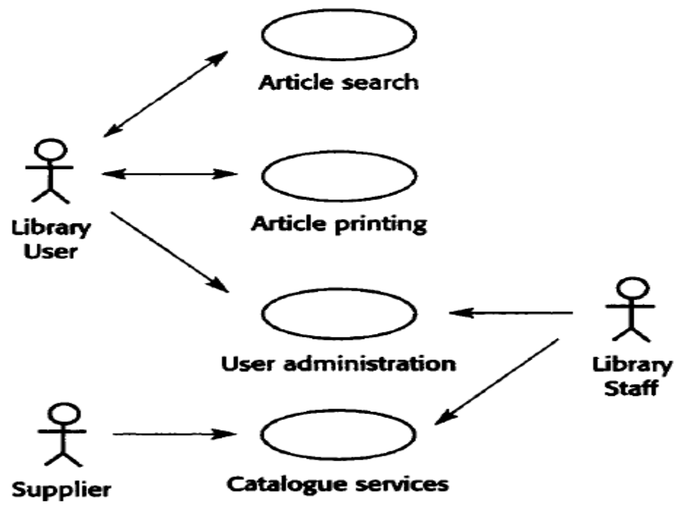
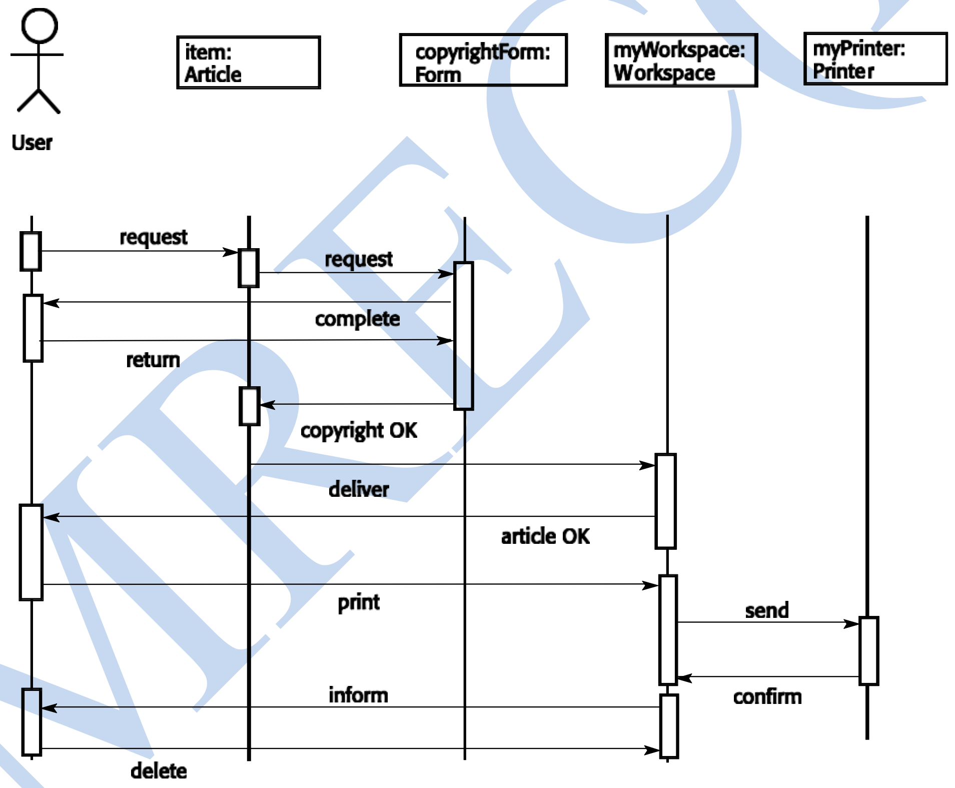


Figure B, Use cases for the library system



Figure, System Interactions for Article printing sequence

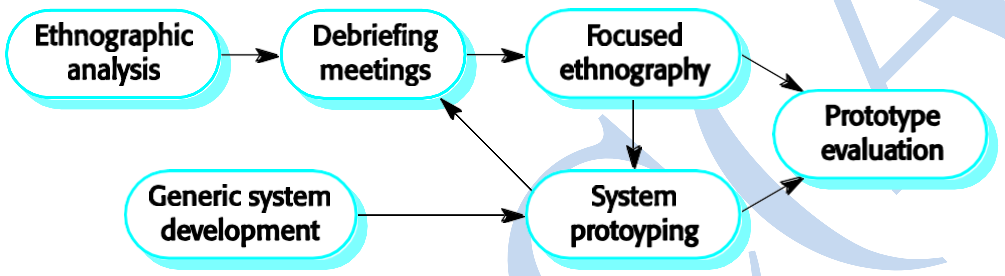
As an illustration of this Figure, shows the interactions involved in using LIBSYS for downloading and printing an article. In the above Figure , there are four objects of classes-Article, Form, Workspace and Printer-involved in this interaction. The sequence of actions is from top to bottom, and the labels on the arrows between the actors and objects indicate the names of operations. Essentially, a user request for an article triggers a request for a copyright form. Once the user has completed the form, the article is downloaded and sent to the printer. Once printing is complete, the article is deleted from the LIBSYS workspace.

2. ETHNOGRAPHY:

- A social scientists spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Focused ethnography:

- Developed in a project studying the air traffic control process
- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis.
- The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.



Figure, Ethnography and prototyping for requirements analysis

Scope of ethnography:

- Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work.
- Requirements that are derived from cooperation and awareness of other people's activities.

REQUIREMENTS VALIDATION:-

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking:

- ***Validity***: Does the system provide the functions which best support the customer's needs?
- ***Consistency***: Are there any requirements conflicts?
- ***Completeness***: Are all functions required by the customer included?
- ***Realism***: Can the requirements be implemented given available budget and technology
- ***Verifiability***: Can the requirements be checked?

Requirements validation techniques

- Requirements reviews
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements. Covered in Chapter 17.
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements reviews:

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.

- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks:

- ***Verifiability*:** Is the requirement realistically testable?
- ***Comprehensibility*:** Is the requirement properly understood?
- ***Traceability*:** Is the origin of the requirement clearly stated?
- ***Adaptability*:** Can the requirement be changed without a large impact on other requirements?

REQUIREMENTS MANAGEMENT:-

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements are inevitably incomplete and inconsistent
  - New requirements emerge during the process as business needs change and a better understanding of the system is developed;
  - Different viewpoints have different requirements and these are often contradictory.

Requirements change

- The priority of requirements from different viewpoints changes during the development process.
- System customers may specify requirements from a business perspective that conflict with end-user requirements.
- The business and technical environment of the system changes during its development.

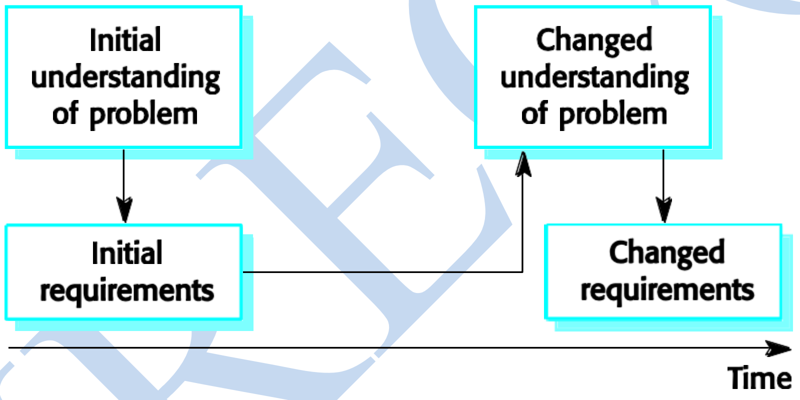

Initial understanding of problem
Changed understanding of problem
Initial requirements
Changed requirements
Time

Figure , Requirements evolution

1). Enduring and volatile requirements:

- ***Enduring requirements***: Stable requirements derived from the core activity of the customer organisation. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models
- ***Volatile requirements***: Requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy

Requirements classification:

Requirement Type	Description
Mutable requirements	Requirements that change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.
Consequential requirements	Requirements that result from the introduction of the computer system. Introducing the computer system may change the organisations processes and open up new ways of working which generate new system requirements

Compatibility requirements	Requirements that depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

2). Requirements management planning:

- During the requirements engineering process, you have to plan:
  - Requirements identification
    - How requirements are individually identified;
  - A change management process
    - The process followed when analysing a requirements change;
  - Traceability policies
    - The amount of information about requirements relationships that is maintained;
  - CASE tool support
    - The tool support required to help manage requirements change;

Traceability:
Traceability is concerned with the relationships between requirements, their sources and the system design

- Source traceability
  - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
  - Links between dependent requirements;
- Design traceability - Links from the requirements to the design;

CASE tool support:

- Requirements storage
  - Requirements should be managed in a secure, managed data store.
- Change management
  - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management

– Automated retrieval of the links between requirements.

3). Requirements change management:

- Should apply to all proposed changes to the requirements.
- Principal stages
  - Problem analysis. Discuss requirements problem and propose change;
  - Change analysis and costing. Assess effects of change on other requirements;
  - Change implementation. Modify requirements document and other documents to reflect change.

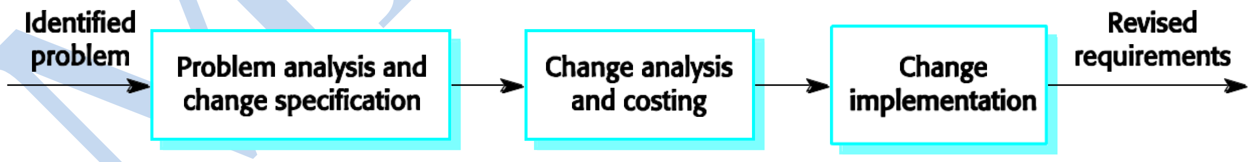

Identified problem
Problem analysis and change specification
Change analysis and costing
Change implementation
Revised requirements

Figure , Requirements Change management

Module – III
Phases of Software Engineering

SYSTEM MODELLING

- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.
- Different models present the system from different perspectives
  - An external perspective, where the context or environment of the system is modelled
  - Behavioural perspective showing the behaviour of the system;
  - Structural perspective showing the system or data architecture.

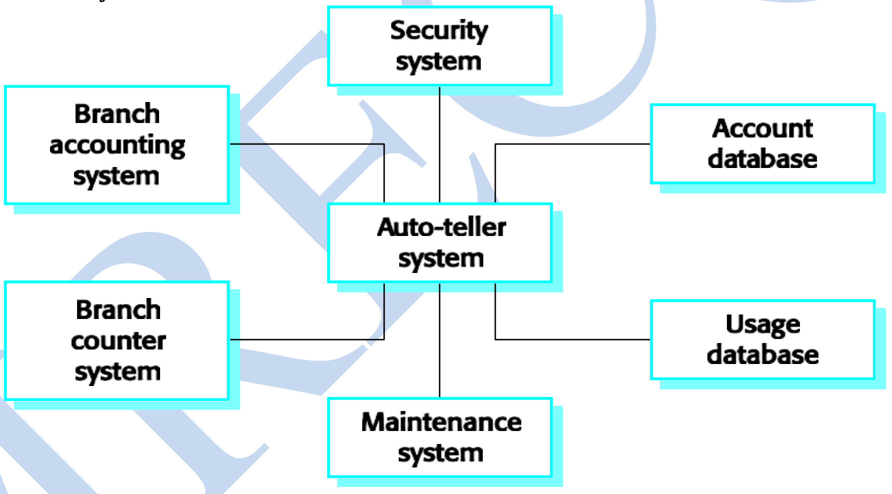
Model types

- Data processing model showing how the data is processed at different stages.
- Composition model showing how entities are composed of other entities.
- Architectural model showing principal sub-systems.
- Classification model showing how entities have common characteristics.
- Stimulus/response model showing the system's reaction to events.

CONTEXT MODELS:-

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

The context of an ATM system:



From Figure, we see that each ATM is connected to an account database, a local branch accounting system, a security system and a system to support machine maintenance. The system is also connected to a usage database that monitors how the network of ATMs is used and to a local branch counter system. This counter system provides services such as backup and printing. These, therefore, need not be included in the ATM system itself.

Process models:

- Process models show the overall process and the processes that are supported by the system.
- Data flow models may be used to show the processes and the flow of information from one process to another.

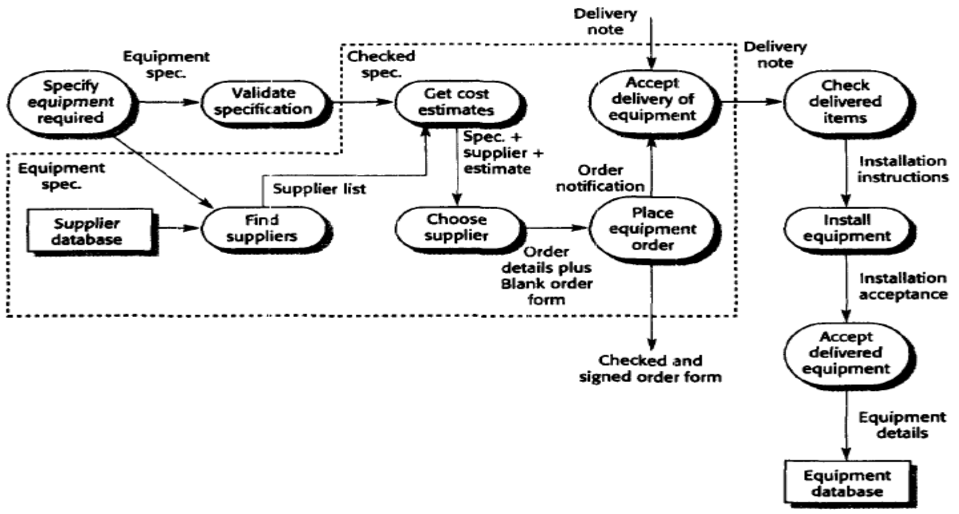


Figure , Process model of equipment procurement

Figure , illustrates a process model for the process of procuring equipment in an organisation. This involves specifying the equipment required, finding and choosing suppliers, ordering the equipment, taking delivery of the equipment and testing it after delivery. When specifying computer support for this process, you have to decide which of these activities will actually be supported. The other activities are outside the boundary of the system. In Figure , the dotted line encloses the activities that are within the system boundary.

BEHAVIOURAL MODELS:-

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
  - i. Data processing models that show how data is processed as it moves through the system;
  - ii.State machine models that show the systems response to events.
- These models show different perspectives so both of them are required to describe the system's behaviour.

i) .**Data-processing models:**

- Data flow diagrams (DFDs) may be used to model the system's data processing.
- These show the processing steps as data flows through a system.
- DFDs are an intrinsic part of many analysis methods.
- Simple and intuitive notation that customers can understand.
- Show end-to-end processing of data.

Order processing DFD:

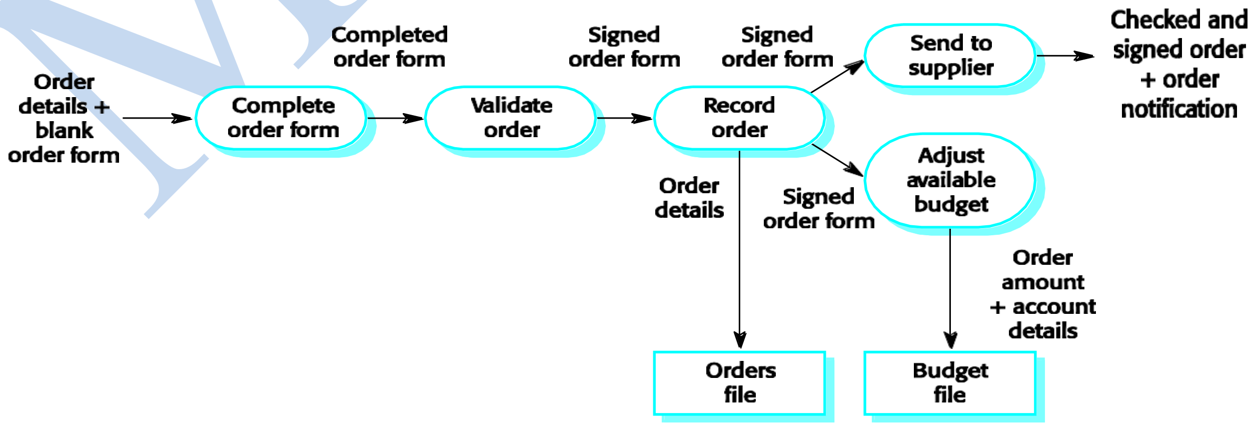


Figure , Data flow diagram of order processing

- DFDs model the system from a functional perspective.
- Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.
- Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

ii.State machine models:

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

Statecharts:

- Allow the decomposition of a model into sub-models (see following slide).
- A brief description of the actions is included following the 'do' in each state.
- Can be complemented by tables describing the states and the stimuli.

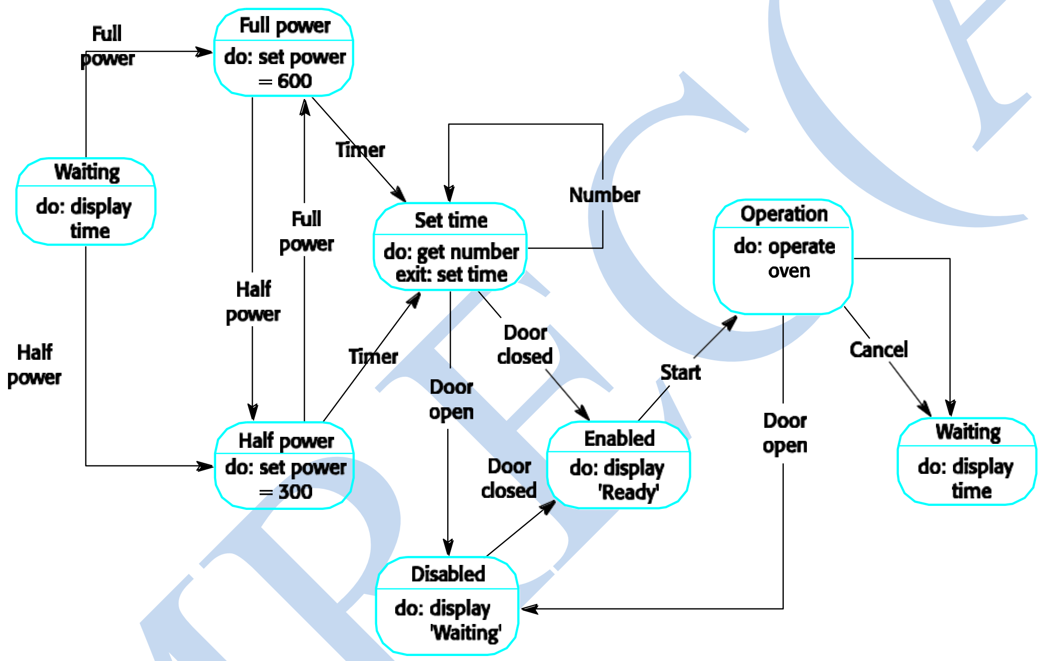


Figure ,State machine model of a simple Microwave oven

Microwave oven state description:

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.

Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Microwave oven stimuli:

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

DATA MODELS:-

- Used to describe the logical structure of data processed by the system.
- An entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes
- Widely used in database design. Can readily be implemented using relational databases.
- No specific notation provided in the UML but objects and associations can be used.

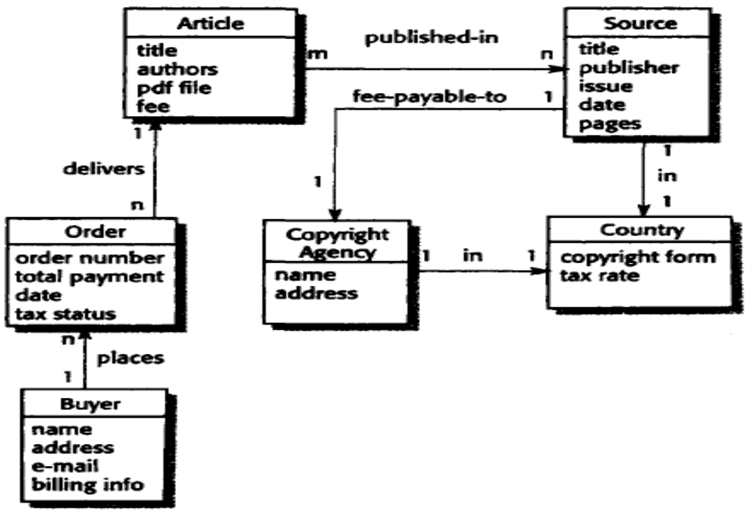


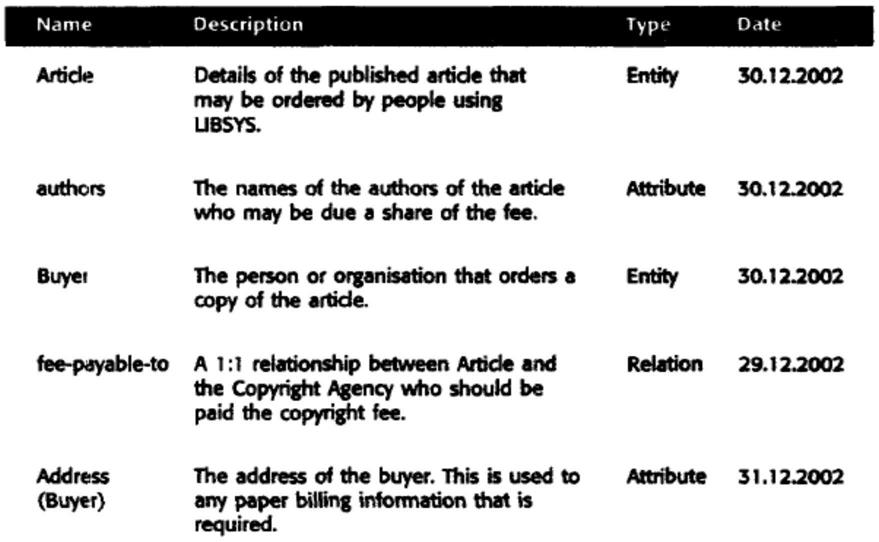
Figure , Semantic data model for the LIBSYS system

Figure , is an example of a data model that is part of the library system LIBSYS introduced in earlier chapters. Recall that LIBSYS is designed to deliver copies of copyrighted articles that have been published in magazines and journals and to collect payments for these articles. Therefore, the data model must include information about the article, the copyright holder and the buyer of the article. I have assumed that payments for articles are not made directly but through national copyright agencies.

Figure , shows that an Article has attributes representing the title, the authors, the name of the PDF file of the article and the fee payable. This is linked to the Source, where the article was published, and to the Copyright Agency for the country of publication. Both Copyright Agency and Source are linked to Country. The country of publication is important because copyright laws vary by country. The diagram also shows that Buyers place Orders for Articles.

Data dictionaries

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included.
- Advantages
    - Support name management and avoid duplication;
    - Store of organisational knowledge linking analysis, design and implementation;
- Many CASE workbenches support data dictionaries.



Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a copy of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

Figure , Examples of data dictionary entries

OBJECT MODELS:-

- Object models describe the system in terms of object classes and their associations.
- An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object.
- Various object models may be produced
    - Inheritance models;
    - Aggregation models;
    - Interaction models.
- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

i). Inheritance models:

- Organise the domain object classes into a hierarchy.
- Classes at the top of the hierarchy reflect the common features of all classes.
- Object classes inherit their attributes and services from one or more super-classes. these may then be specialised as necessary.
- Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

Object models and the UML:

- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
    - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
    - Relationships between object classes (known as associations) are shown as lines linking objects;
    - Inheritance is referred to as generalisation and is shown 'upwards' rather than 'downwards' in a hierarchy.

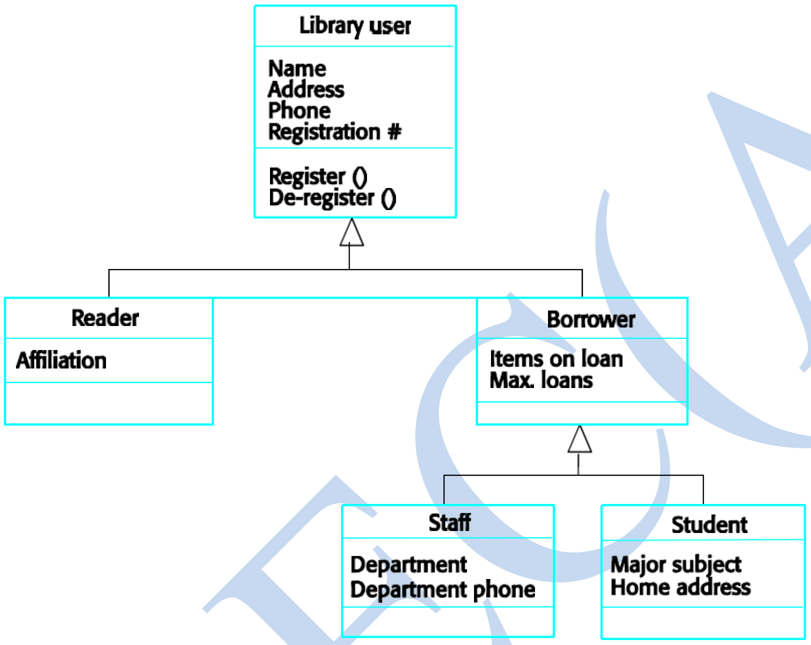


Figure , User class hierarchy

Multiple inheritance:

- Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes.
- This can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics.
- Multiple inheritance makes class hierarchy reorganisation more complex.

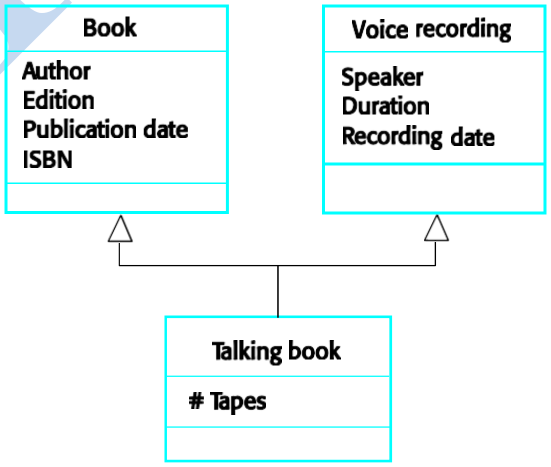


Figure , Multiple inheritance

ii). Object aggregation:

- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.

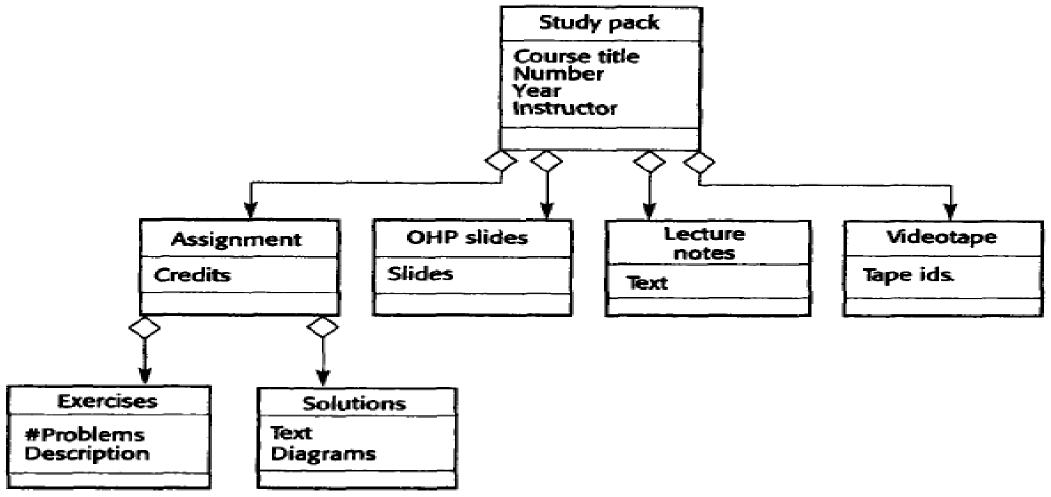


Figure , Aggregation object representing a course

iii). Object behaviour modelling

- A behavioural model shows the interactions between objects to produce some particular system behaviour that is specified as a use-case.
- Sequence diagrams (or collaboration diagrams) in the UML are used to model interaction between objects.

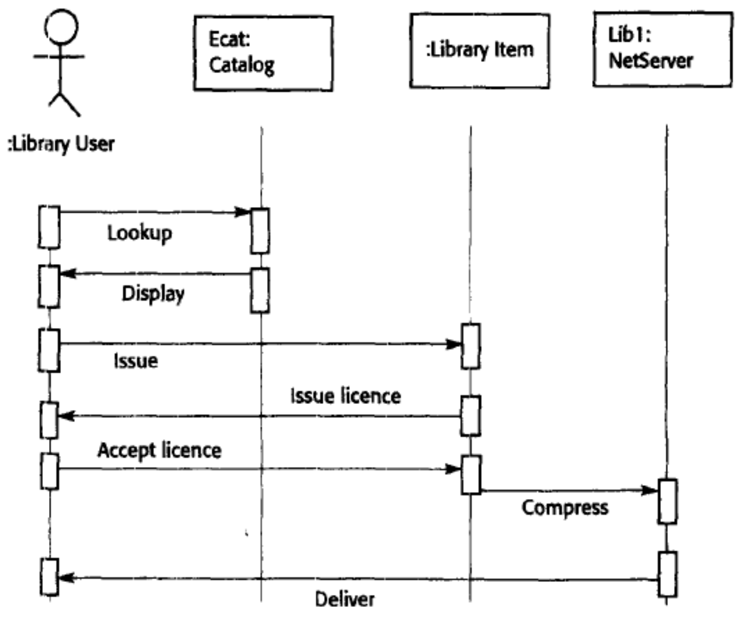


Figure , The issue of electronic items

In a sequence diagram, objects and actors are aligned along the top of the diagram. Labelled arrows indicate operations; the sequence of operations is from top to bottom. In this scenario, the library user accesses the catalogue to see whether the item required is available electronically; if it is, the user requests the electronic issue of that item. For copyright reasons, this must be licensed so there is a transaction between the item and the user where the license is agreed. The item to be issued is then sent to a network server object for compression before being sent to the library user.

STRUCTURED METHODS:-

- Structured methods incorporate system modelling as an inherent part of the method.
- Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models.
- CASE tools support system modelling as part of a structured method.

Structured Method weaknesses:

- They do not model non-functional system requirements.
- They do not usually include information about whether a method is appropriate for a given problem.
- The may produce too much documentation.
- The system models are sometimes too detailed and difficult for users to understand.

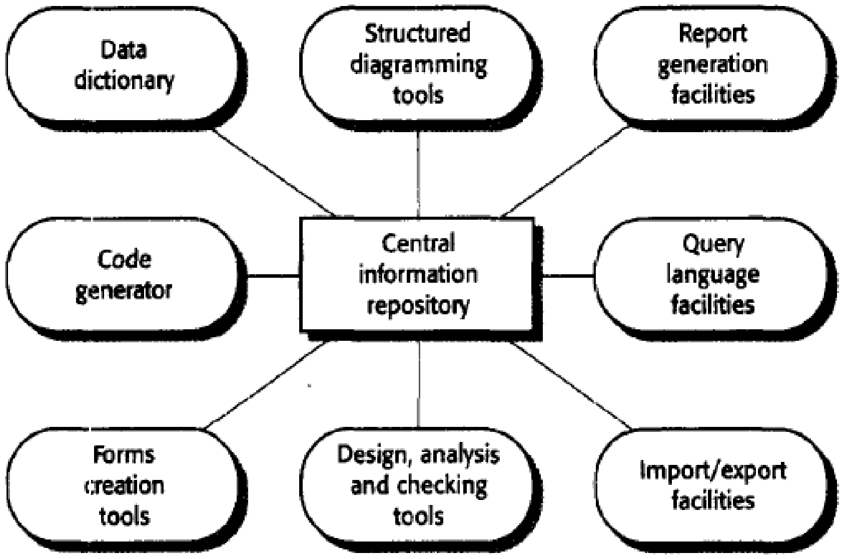

Data dictionary
Structured diagramming tools
Report generation facilities
Code generator
Central information repository
Query language facilities
Forms creation tools
Design, analysis and checking tools
Import/export facilities

Figure , The components of a CASE tool for structured method support

Comprehensive method support tools, as illustrated in Figure , normally include:

1. ***Diagram editors*** used to create object models, data models, behavioural models, and so on. These editors are not just drawing tools but are aware of the types of entities in the diagram. They capture information about these entities and save this information in the central repository.

2. ***Design analysis and checking tools*** that process; the design and report on error and anomalies. These may be integrated with the editing system so that user errors are trapped at an early stage in the process.

3. ***Repository query languages*** that allow the designer to find designs and associated design information in the repository.

4. **A *data dictionary*** that maintains information about the entities used in a system design.

5. ***Report definition and generation tools*** that take information from the central store and automatically generate system documentation.

6. ***Forms definition tools*** that allow screen and document formats to be specified.

7. ***Import/export facilities*** that allow the interchange of information from the central repository with other development tools.

8. ***Code generators*** that generate code or code skeletons automatically from the design captured in the central store.

DESIGN ENGINEERING:-

Design engineering encompasses the **set of principals, concepts, and practices** that lead to the development of a high- quality system or product.

- ✓ Design principles establish an overriding philosophy that guides the designer in the work that is performed.
- ✓ Design concepts must be understood before the mechanics of design practice are applied and
- ✓ Design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

What is design:

Design is what virtually every engineer wants to do. It is the place where creativity rules –customer's requirements, business needs, and technical considerations all come together in the formulation of a product or a system. Design creates a representation or model of the software, but unlike the analysis model, the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Why is it important:

Design allows a software engineer to model the system or product that Is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end – users become involved in large numbers. Design is the place where software quality is established.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence. Another **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

DESIGN PROCESS AND DESIGN QUALITY:-

Design process:

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Goals of design:

McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design.

- ➢ The design must implement all ofthe explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- ➢ The design must be a readable, understandable guide for those who generate code and forthose who test and subsequently support the software.
- ➢ The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality guidelines:

In order to evaluate the quality of a design representation we must establish technical criteria for good design. These are the following guidelines:

1. A design should exhibit an architecture that
   - **a.** has been created using recognizable architectural styles or patterns
   - **b.** is composed of components that exhibit good design characteristics and
   - **c.** can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representation of data, architecture, interfaces and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interface that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven byinformation obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communication its meaning.

These design guidelines are not achieved by chance. Design engineering encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

Quality attributes:

The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities ofthe program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, and the mean – time –to- failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by processing speed, response time, resource consumption, throughput, and efficiency
- ***Supportability c****ombines the ability to extend the program (extensibility), adaptability, serviceability- these three attributes represent a more common term maintainability

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security. Another may demand performance with particular emphasis on processing speed. A third might focus on reliability.

DESIGN CONCEPTS:-

M.A Jackson once said:"The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."

I. Abstraction: Many levels of abstraction are there.

- ✓ At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- ✓ At lower levels of abstraction, a more detailed description of the solution is provided.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of procedural abstraction implies these functions, but specific details are suppressed.
A ***data abstraction*** is a named collection of data that describes a data object.
In the context of the procedural abstraction *open*, we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing operation, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

II. Architecture:

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

One **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

The architectural design can be represented using one or more of a number of different models. ***Structured models*** represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function external events.

Process models focus on the design of the business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

III. Patterns:

Brad Appleton defines a ***design pattern*** in the following manner: "a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns." Stated in another way, a design pattern describes a design structure that solves a particular design within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.
The intent of each design pattern is to provide a description that enables a designer to determine

1) Whether the pattern is capable to the current work,
2) Whether the pattern can be reused,
3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

IV. Modularity:

Software architecture and design patterns embody ***modularity*****;** software is divided into separately named and addressable components, sometimes called ***modules*** that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

V. Information Hiding:

The principle of *information hiding* suggests that modules be "characterized by design decision that hides from all others."
Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within software.

VI. Functional Independence:

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. *Functional independence* is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific sub function of requirements and has a simple interface when viewed from other parts of the program structure.

Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information hiding.

A cohesion module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

VII. Refinement:

Stepwise refinement is a top- down design strategy originally proposed by Niklaus wirth. A program is development by successively refining levels of procedural detail. A hierarchy is development by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

VIII. Refactoring :

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

IX. Design classes:

The software team must define a set of design classes that

1. Refine the analysis classes by providing design detail that will enable the classes to be implemented, and
2. Create a new set of design classes that implement a software infrastructure to support the design solution.

Five different types of design classes, each representing a different layer of the design architecture are suggested.

- **User interface classes:** define all abstractions that are necessary for human computer interaction. In many cases, HCL occurs within the context of a *metaphor* and the design classes for the interface may be visual representations of the elements of the metaphor.
- **Business domain classes:** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.

- **Process classes** implement lower – level business abstractions required to fully manage the business domain classes.
- **Persistent classes** represent data stores that will persist beyond the execution of the software.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

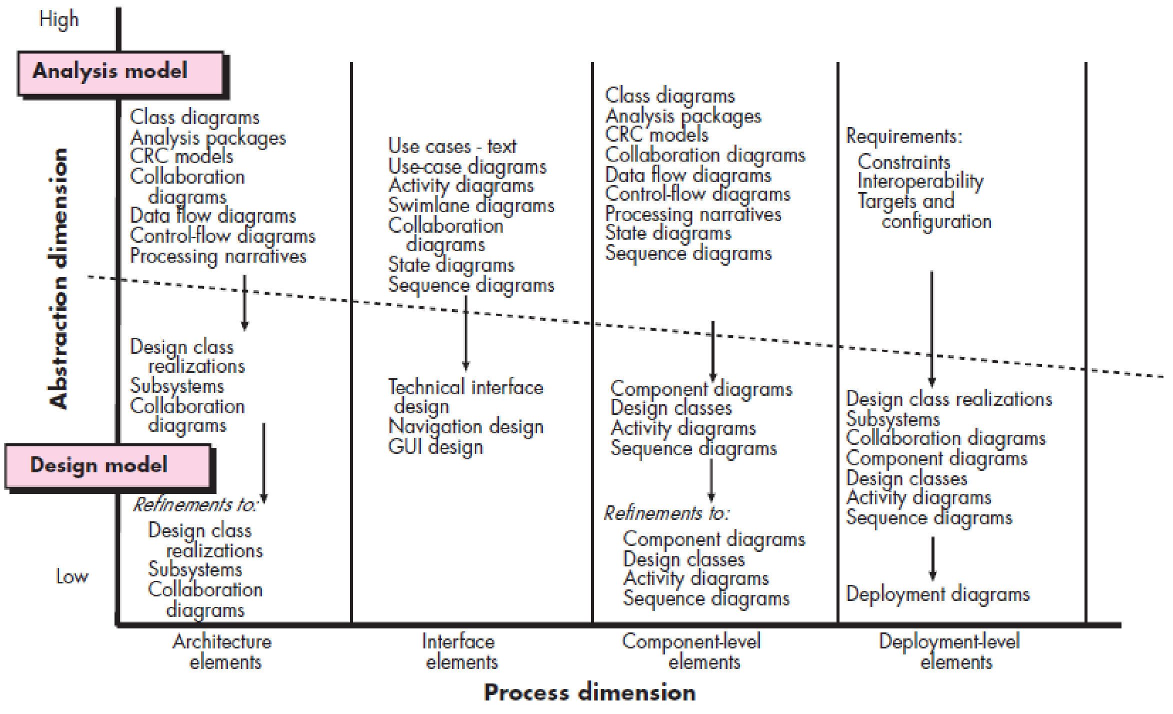
THE DESIGN MODEL:-

- The design model can be viewed into different dimensions.
- The process dimension indicates the evolution of the design model as design tasks are executed as a part of the software process.

The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.
The elements of the design model use many of the same UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as a path of design; more implementation- specific detail is provided, and architectural structure and style, components that reside within the architecture, and the interface between the components and with the outside world are all emphasized.

It is important to mention however, that model elements noted along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model us usually delayed until the design has been fully developed.

FIGURE 8.4 Dimensions of the design model



i. **Data design elements:**

Data design sometimes referred to as data architecting creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
The structure of data has always been an important part of software design.

- ✓ At the **program component level**, the design of data structures and the associated algorithms required to manipulate them is essential to the criterion of high-quality applications.

- ✓ At **the application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- ✓ At the **business leve**l, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

ii. Architectural design elements:

The *architectural design* for software is the equivalent to the floor plan of a house.
The architectural model is derived from three sources.

1) Information about the application domain for the software to be built.
2) Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
3) The availability of architectural patterns

iii. Interface design elements:

The *interface design* for software is the equivalent to a set of detailed drawings for the doors, windows, and external utilities of a house.

The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are 3 important elements of interface design:

1) The user interface(UI);
2) External interfaces to other systems, devices, networks, or other produces or consumers of information; and
3) Internal interfaces between various design components.

These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

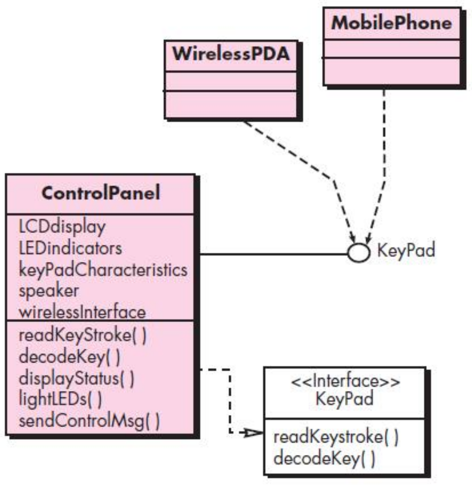
UI design is a major software engineering action.

The design of a UI incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

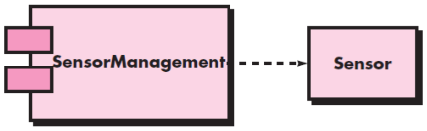
UML defines an *interface* in the following manner:"an interface is a specifier for the externally-visible operations of a class, component, or other classifier without specification of internal structure."

FIGURE 8.5
Interface representation for Control-Panel



iv. **Component- level design elements:** The component-level design for software is equivalent to a set of detailed drawings.

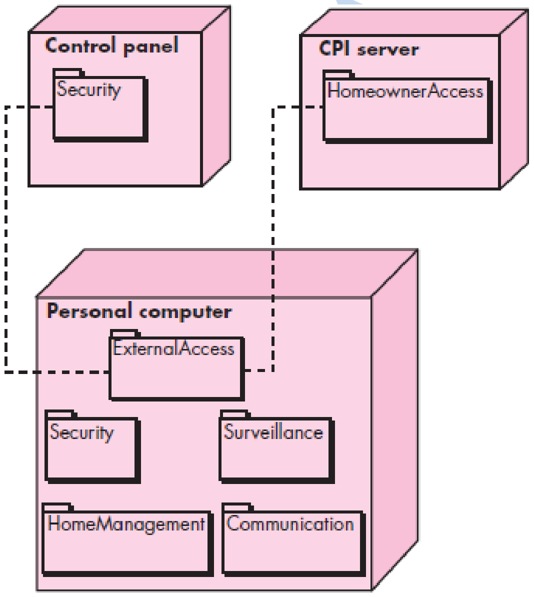
The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.



Figure, A UML component diagram

v. **Deployment-level design elements:** Deployment-level design elements indicated how software functionality and subsystems will be allocated within the physical computing environment that will support the software

A UML deployment diagram



ARCHITECTURAL DESIGN

SOFTWARE ARCHITECTURE:-

What Is Architecture?

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers

- the architectural style that the system will take,
- the structure and properties of the components that constitute the system, and
- the interrelationships that occur among all architectural components of a system.

The architecture is a representation that enables a software engineer to

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, (3) reducing the risks associated with the construction of the software.

The design of software architecture considers two levels of the design pyramid

- data design
- architectural design.

✓ Data design enables us to represent the data component of the architecture.

✓ Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?

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

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

DATA DESIGN:-

The data design activity translates data objects as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

➢ At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.

➢ At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.

➢ At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

I. Data design at the Architectural Level:

The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional.

To solve this challenge, the business IT community has developed ***data mining*** techniques, also called ***knowledge discovery in databases*** (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a ***data warehouse***, adds an additional layer to the data architecture. a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business.

II. Data design at the Component Level:

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. The following set of principles for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low-level data design decisions should be deferred until late in the design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

ARCHITECTURAL STYLES and Patterns :-

Architectural Styles:

The builder has used an ***architectural style*** as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod).

The software that is built for computer-based systems also exhibits one of many architectural styles.

Each style describes a system category that encompasses

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

(2) A set of ***connectors*** that enable "communication, coordinations and cooperation" among components;

(3) ***Constraints*** that define how components can be integrated to form the system; and

(4) ***Semantic models*** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

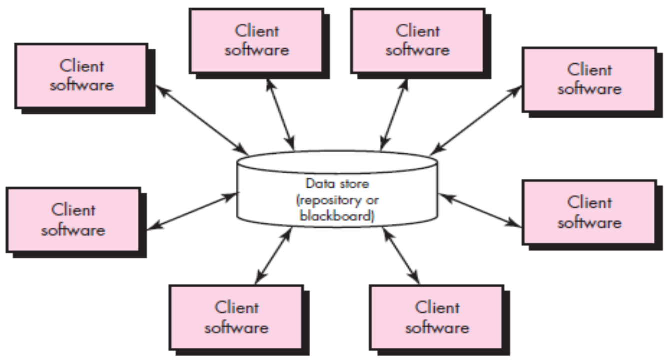
1).A Brief Taxonomy of Styles and Patterns

Data-centered architectures:

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A variation on this approach transforms the repository into a "blackboard" that sends notification to client software when data of interest to the client changes

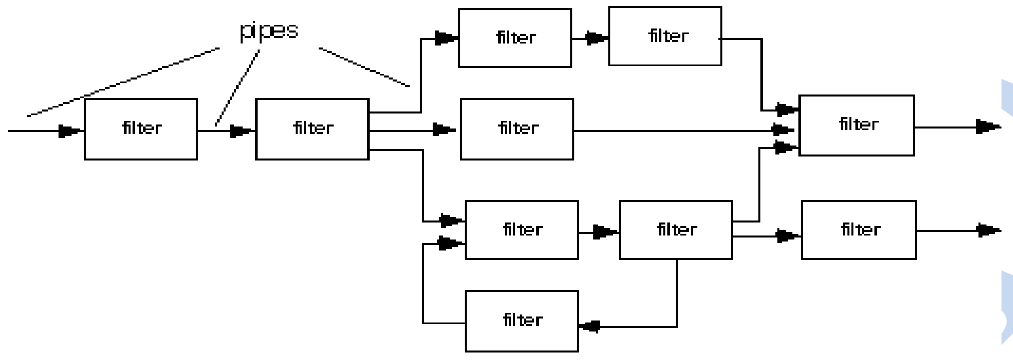
Data-centered architectures promote ***integrability***. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism

FIGURE 9.1
Data-centered architecture

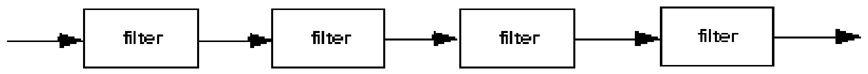


Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A ***pipe and filter pattern*** has a set of components, called ***filters***, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output of a specified form.

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



(a) pipes and filters

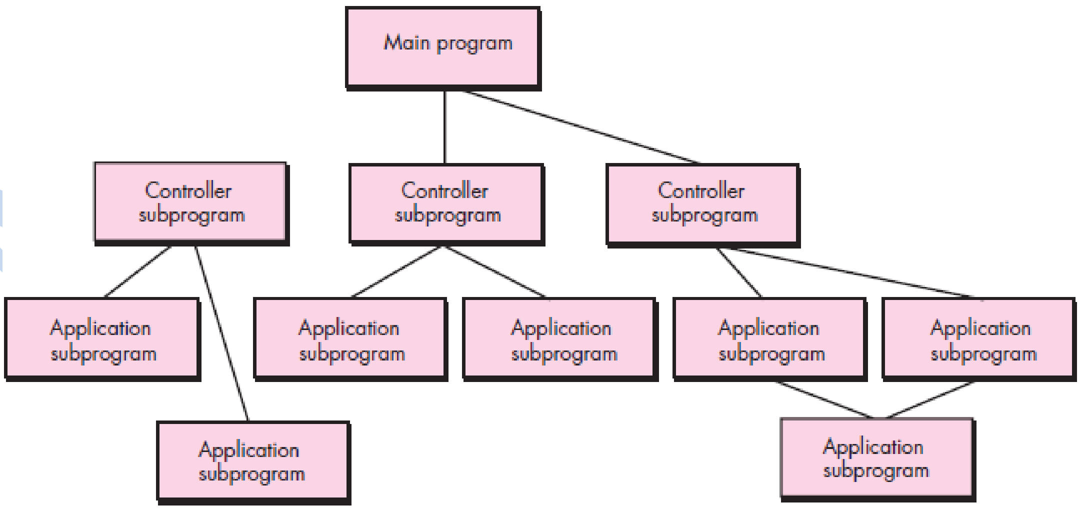


(b) batch sequential

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

- ***Main program/subprogram architectures***. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure illustrates an architecture of this type.
- ***Remote procedure call architectures***. The components of a main program/ subprogram architecture are distributed across multiple computers on a network

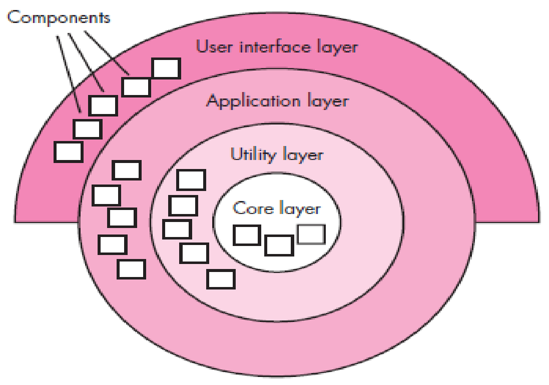
FIGURE 9.3 Main program/subprogram architecture



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

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 14.3. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

FIGURE 9.4
Layered architecture



2). Architectural Patterns:-

An ***architectural pattern,*** like an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.

(2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

(3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

The architectural patterns for software define a specific approach for handling some behavioral characteristics of the system

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

- *operating system process management* pattern
- *task scheduler* pattern

Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:

- a ***database management system*** pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an ***application level persistence*** pattern that builds persistence features into the application architecture

Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment

- A ***broker*** acts as a 'middle-man' between the client component and a server component.

3). Organization and Refinement:

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

Control.

✓ How is control managed within the architecture?
✓ Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
✓ How do components transfer control within the system?
✓ How is control shared among components?

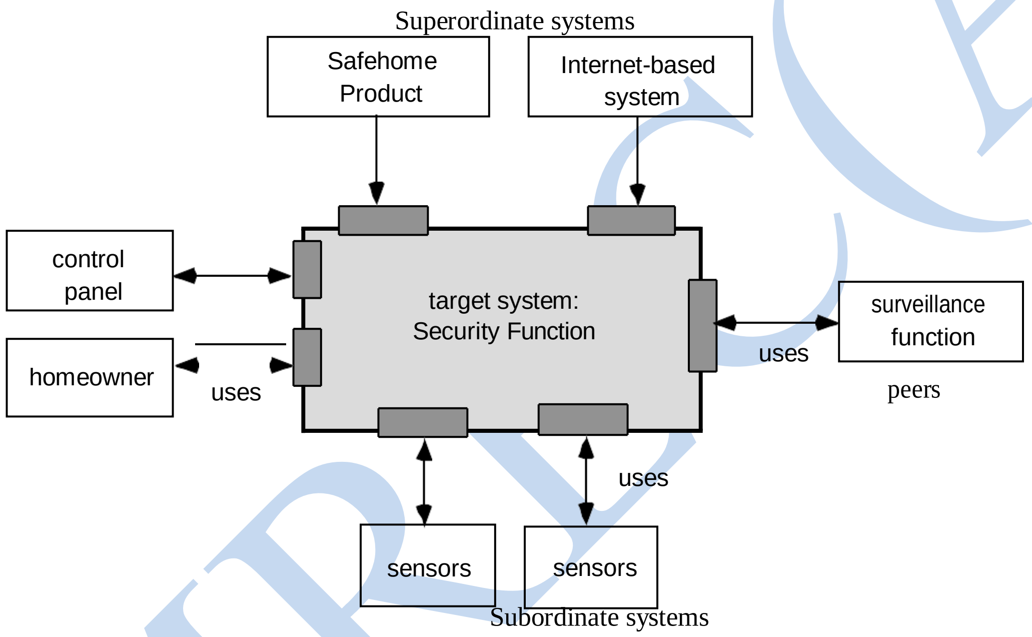
Data.

- ✓ How are data communicated between components?
- ✓ Is the flow of data continuous, or are data objects passed to the system sporadically?
- ✓ What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)?
- ✓ Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?
- ✓ How do functional components interact with data components?
- ✓ Are data components *passive* or *active* (i.e., does the data component actively interact withother components in the system)? How do data and control interact within the system?

ARCHITECTURAL DESIGN:-

I .Representing the System in Context:

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in the figure



Superordinate systems – those systems that use the target system as part of some higher level processing scheme.

Subordinate systems - those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Peer-level systems - those systems that interact on a peer-to-peer basis

Actors -those entities that interact with the target system by producing or consuming information that is necessary for requisite processing

II .Defining Archetypes:

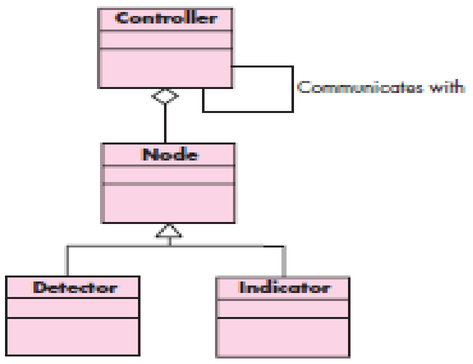
An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system. In general, a relative small set of archetypes is required to design even relatively complex systems.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the analysis model. In safe home security function, the following are the archetypes:

- **Node:** Represent a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors, and (2) a variety of alarm indicators.

- **Detector:** An abstraction that encompasses all sensing equipment that feeds information into the target system
- **Indicator:** An abstraction that represents all mechanisms for indication that an alarm condition is occurring.
- **Controller:** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

FIGURE 9.7
UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].



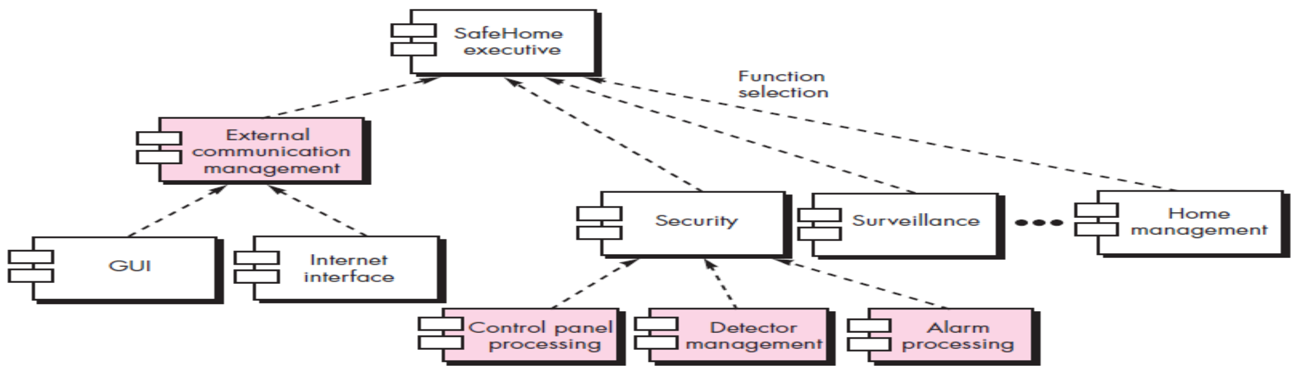
III .Refining the Architecture into Components:

As the architecture is refined into components, the structure of the system begins to emerge. The architectural designer begins with the classes that were described as part of the analysis model. These analysis classes represent entities within the application domain that must be addressed within the software architecture. Hence, the application domain is one source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application domain.

For eg: memory management components, communication components database components, and task management components are often integrated into the software architecture.
In the *safeHome* security function example, we might define the set of top-level components that address the following functionality:

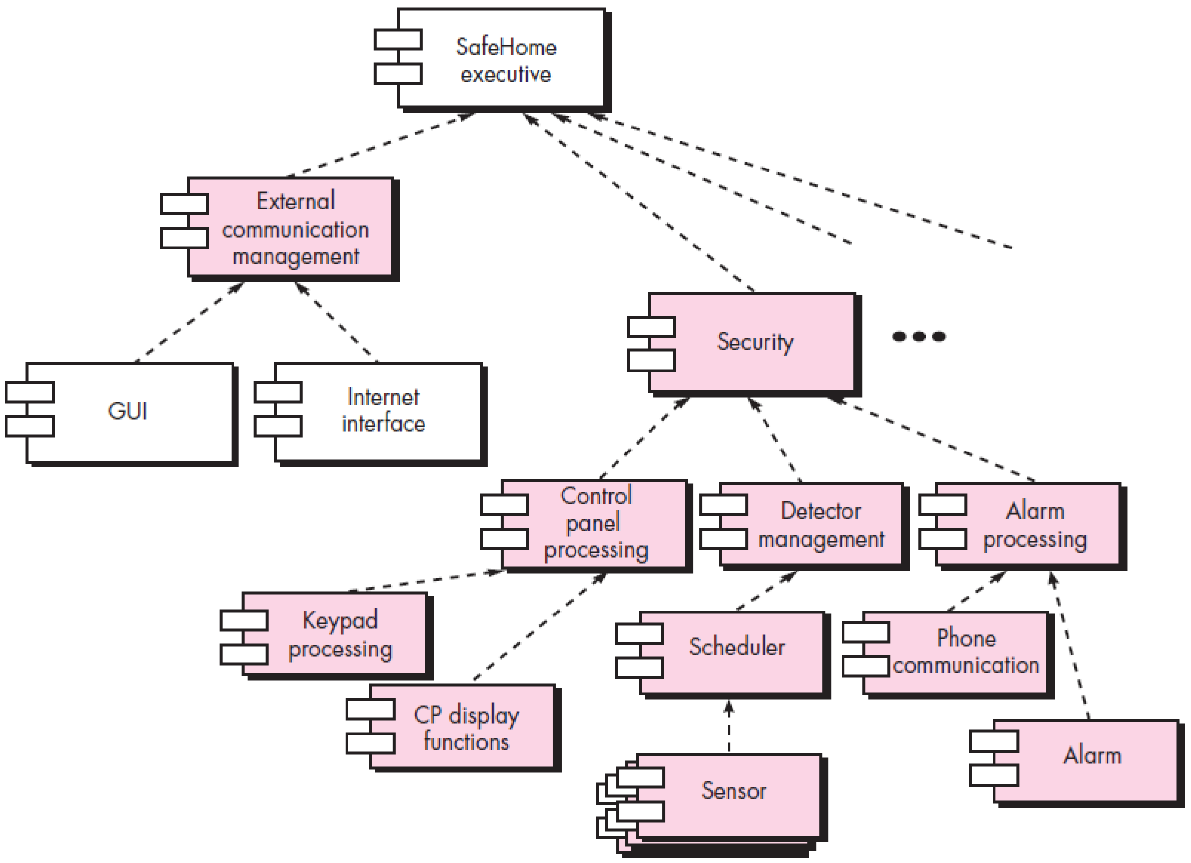
- *External communication management*- coordinates communication of the security function with external entities
- *Control panel processing*- manages all control panel functionality.
- *Detector management*- coordinates access to all detectors attached to the system.
- *Alarm processing*- verifies and acts on all alarm conditions.

Design classes would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design.



Figure, Overall architectural structure for *SafeHome* with top-level components

IV **.Describing Instantiations of the System:** An actual instantiation of the architecture means the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.



Figure, An instantiation of the security function with component elaboration

Module –IV
Test Strategies

Methods of Testing:

A strategic Approach for Software testing:-

Software system testing is a method of verification involving systematically executing software to detect errors.
Glen Myers defines testing as "a process of executing a program with an invent of finding an error". Software testing plays a critical role in the software quality. The following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy should provide guidance for the practitioner and a set of milestones for the manager

i) Verification and Validation:

* Verification refers to set of activities that ensure that software correctly implements a specific function

Example:

Verification: Are we building the product right?

* Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements

Example:

Validation: Are we building the right product?

* Verification and validation includes a wide variety of SQA activities that encompass,

- Formal Technical Reviews
- Quality and Configuration audits
- Performance Monitoring
- Simulation
- Feasibility Study
- Documentation Review
- Database Review
- Analysis Algorithm
- Development Testing
- Usability Testing
- Qualification Testing
- Installation Testing

ii) Organizing for Software Testing:

The software engineer creates a computer program, its documentation, and related data structures. Like any builder, the software engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) *destructive*. So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than uncovering errors. Unfortunately, errors will be present. And, if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that can be erroneously inferred from the preceeding discussion:
(1) that the developer of software should do no testing at all,
(2) that the software should be "tossed over the wall" to strangers who will test it mercilessly,
(3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

The software developer is always responsible for testing the individual units (modules) of the program ensuring that each performs the function for which it was designed. In many cases, the developer also conducts ***integration testing***

A testing step that leads to construction of the complete program structure. Only after the software architecture is complete, does an independent test group (ITG) become involved?

The role of an ITG is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise present. After all, personnel in the ITG team are paid to find errors.

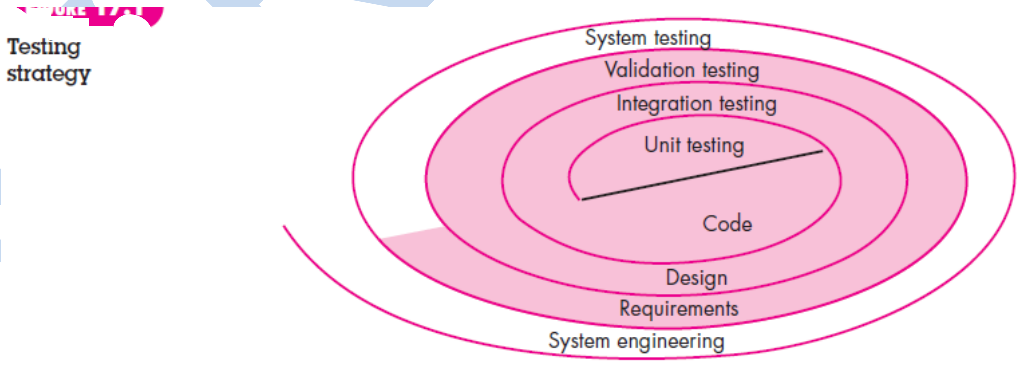
However, the software developer does not turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

The ITG is part of the software development project team in the sense that it becomes involved during the specification process and stays involved (planning and specifying test procedures) throughout a large project.

However, in many cases the ITG reports to the SQA organization, there by achieving a degree of independence that might not be possible if it were a part of the software development organization.

iii) Software testing strategy for conventional software architecture:

* A strategy for software testing may be viewed in the context of the spiral as shown



* Unit testing begins at the vortex of the spiral and concentrates on unit [i.e. components] of the software as implemented in the source code

* Taking another turn by moving along the spiral to integrate testing which focus on design and the construction of software architecture

* Next turn we encounter validation testing which validate requirements established as part of software requirements

analysis against software that has been constructed

* Finally we arrive at system testing, where the software and other system elements are tested as whole

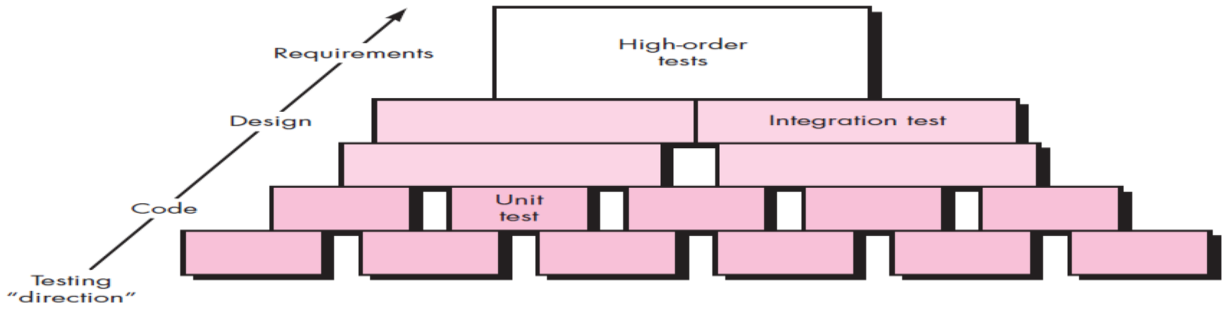
<u>iv) Software Testing Steps:</u>

(i) Unit Testing:

* Initially tests focus on each component individually ensuring that it functions properly as a unit

* Unit testing uses testing techniques heavily, to ensure complete coverage and maximum error detection in components control structure

* The components must be assembled (Or) Integrated into form complete software packages



(ii) Integration Testing:

* It address the issues associated with the dual problems of verification and program construction

* Validation criteria [established during requirements analysis] must be evaluated

* Validation testing provides final assurance that software meets all functional, behavioral and performance requirements

* After the software has been integrated [constructed] a set of high – order tests are conducted

(iii) High Order Testing:

* It falls outside the boundary of the software engineering

* Once software is validated it must be combined with other system elements [ex: hardware, people, and software]

* System testing verifies that all elements mesh properly and that overall system function / performance is achieved

v)Criteria for Completion of Testing

The completion criteria are what we use to determine if we can stop the testing or if we have to go on to reach the objective of the testing.

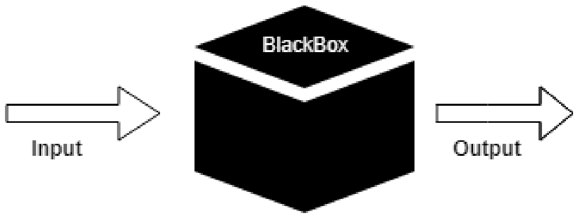
The completion criteria are derived from the strategy and should be based on a risk analysis; the higher the risk, the stricter the completion criteria; the lower the risk the less demanding and specific the completion criteria. It is quite significant step to decide up front which completion criteria should be fulfilled before the test may be stopped.

The completion criteria guide the specification of the test and the selection of different techniques for test case design. These techniques are exploited to provide the test cases that satisfy the completion criteria.

Software Testing

- Two major categories of software testing
- ❖ Black box testing
- ❖ White box testing

Black box testing:-



Black Box Testing is also known as behavioral, opaque-box, closed-box, specification-based or eye-to-eye testing.

It is a Software Testing method that analyses the functionality of a software/application without knowing much about the internal structure/design of the item that is being tested and compares the input value with the output value.

The main focus in Black Box Testing is on the functionality of the system as a whole. The term 'Behavioral Testing' is also used for Black Box Testing. Behavioral test design is slightly different from the black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged.

Each testing method has its own advantages and disadvantages. There are some bugs that cannot be found using the only black box or only white box technique.

Majority of the applications are tested by Black Box method. We need to cover the majority of test cases so that most of the bugs will get discovered by a Black-Box method.

This testing occurs throughout the software development and Testing Life Cycle i.e in Unit, Integration, System, Acceptance, and Regression Testing stages.

This can be both Functional or Non-Functional.

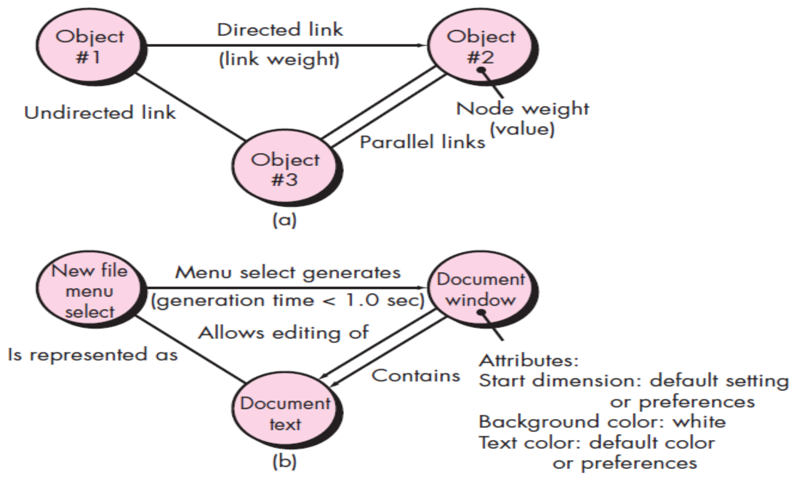
Black Box Testing Techniques

1)Graph based testing method
2) Equivalence partitioning
3) Boundary Value analysis
4) Orthogonal array Testing

1. Graph based testing

This technique of Black box testing involves a graph drawing that depicts the link between the causes (inputs) and the effects (output), which trigger the effects.

This testing utilizes different combinations of output and inputs. It is a helpful technique to understand the software's functional performance, as it visualizes the flow of inputs and outputs in a lively fashion.



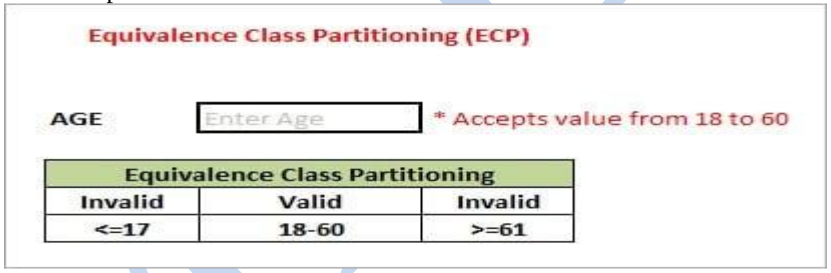
The symbolic representation of a graph is shown in Figure , Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes

2. Equivalence partitioning

This technique is also known as Equivalence Class Partitioning (ECP). In this technique, input values to the system or application are divided into different classes or groups based on its similarity in the outcome.

Hence, instead of using each and every input value we can now use any one value from the group/class to test the outcome. In this way, we can maintain the test coverage while we can reduce a lot of rework and most importantly the time spent.

For Example:



Equivalence Class Partitioning (ECP)

AGE Enter Age * Accepts value from 18 to 60

Equivalence Class Partitioning		
Invalid	Valid	Invalid
<=17	18-60	>=61

Equivalence Partitioning

As present in the above image, an “AGE” text field accepts only the numbers from 18 to 60. There will be three sets of classes or groups.

Two invalid classes will be:

a) Less than or equal to 17.

b) Greater than or equal to 61.

One valid class will be anything between 18 to 60.

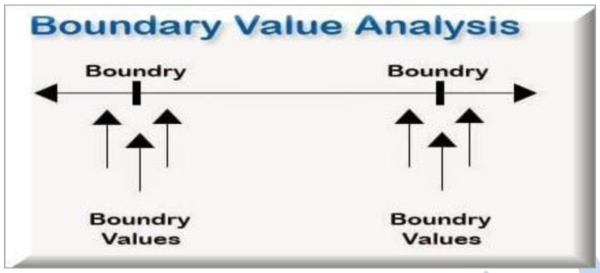
We have thus reduced the test cases to only 3 test cases based on the formed classes thereby covering all the possibilities. So, testing with anyone value from each set of the class is sufficient to test the above scenario.

3. Boundary Value analysis

From the name itself, we can understand that in this technique we focus on the values at boundaries as it is found that many applications have a high amount of issues on the boundaries.

Boundary means the values near the limit where the behavior of the system changes. In boundary value analysis both the valid inputs and invalid inputs are being tested to verify the issues.

For Example:



Boundary Value Analysis

If we want to test a field where values from 1 to 100 should be accepted then we choose the boundary values: 1-1, 1, 1+1, 100-1, 100, and 100+1. Instead of using all the values from 1 to 100, we just use 0, 1, 2, 99, 100, and 101.

4. Orthogonal array Testing

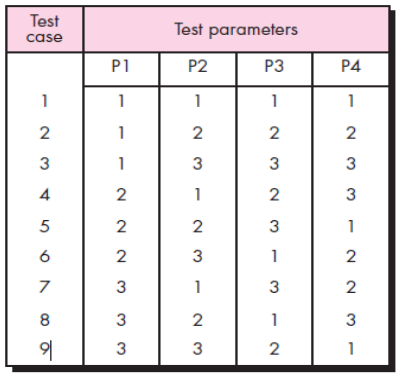
Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.

To problems in which input domain is relatively small but too large for exhaustive testing

Three inputs A,B,C each having three values will require $3^4 = 81$ test cases

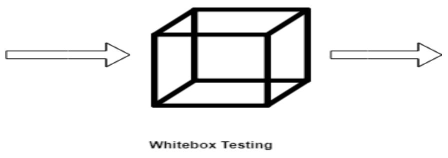
L9 orthogonal testing will reduce the number of test case to 9 as shown below

Example



Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

White Box testing:-



White box testing is a testing technique, that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

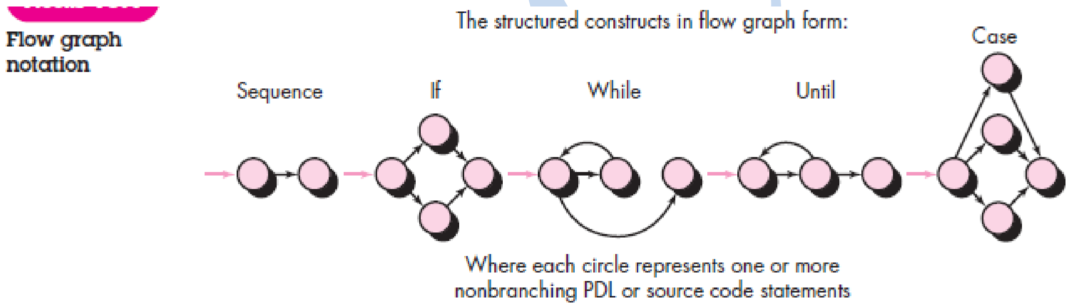
White Box Testing Techniques

1. Basis path testing
2. Control structure testing

Basis path testing:- Proposed by Tom McCabe

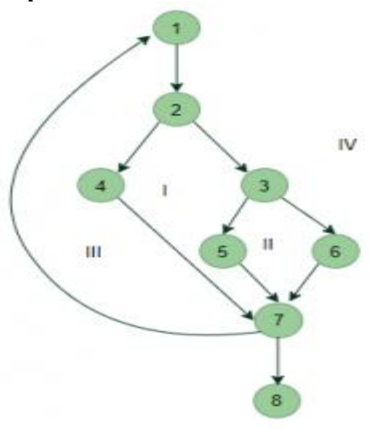
In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.

i)Flow graph notation: It is a directed graph consisting of nodes and edges. Each node represents a sequence of statements, or a decision point. A predicate node is the one that represents a decision point that contains a condition after which the graph splits. Regions are bounded by nodes and edges.


Flow graph notation
The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

ii)Cyclomatic Complexity: It is a measure of the logical complexity of the software and is used to define the number of independent paths. For a graph G, V(G) is its cyclomatic complexity.
Calculating V(G):
1. V(G) = P + 1, where P is the number of predicate nodes in the flow graph
2. V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes
3. V(G) = Number of non-overlapping regions in the graph

Example:



V(G) = 4 (Using any of the above formulae)
No of independent paths = 4

- #P1: 1 – 2 – 4 – 7 – 8
- #P2: 1 – 2 – 3 – 5 – 7 – 8
- #P3: 1 – 2 – 3 – 6 – 7 – 8
- #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

Control structure testing:-

a) Condition Coverage: In this technique, all individual conditions must be covered as shown in the following example:

1. READ X, Y
2. IF(X == 0 || Y == 0)
3. PRINT '0'

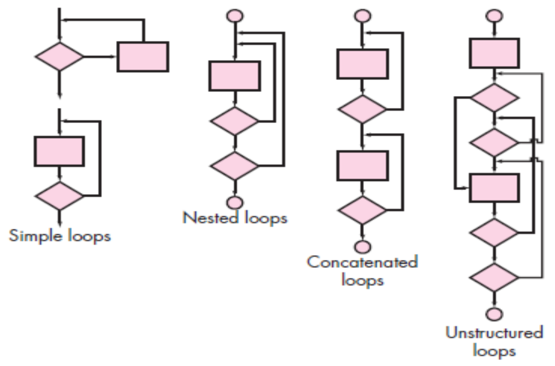
In this example, there are 2 conditions: X == 0 and Y == 0. Now, test these conditions get TRUE and FALSE as their values. One possible example would be:

- #TC1 – X = 0, Y = 55
- #TC2 – X = 5, Y = 0

b)Loop Testing: Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.

1. **Simple loops:** For simple loops of size n, test cases are designed that:
   - Skip the loop entirely
   - Only one pass through the loop
   - 2 passes
   - m passes, where m < n
   - n-1 ans n+1 passes
2. **Nested loops:** For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.
3. **Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each.
   If they're not independent, treat them like nesting.

Classes of Loops



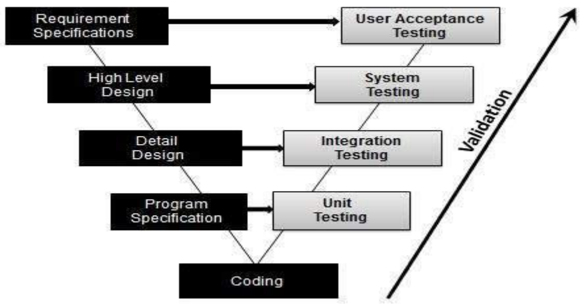
Validation Testing:-

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

Validation Testing - Workflow:

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.



It succeeds when the software functions in a manner that can be reasonably expected by the customer.

i)Validation Test Criteria

validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created. Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery

ii)Configuration Review

An important element of the validation process is a *configuration review*. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an audit

iii)Alpha And Beta Testing

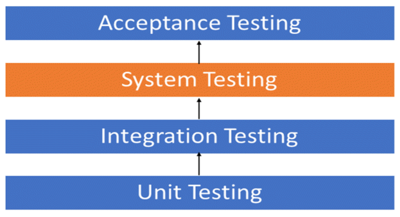
The ***alpha test*** is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment

The ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base

System Testing:-

System Testing means testing the system as a whole. All the modules/components are integrated in order to verify if the system works as expected or not.

System Testing is done after Integration Testing. This plays an important role in delivering a high-quality product.



Its primary purpose is to test the complete software.

1)**Recovery Testing**:- is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

2.**Security Testing** : To make sure that the system does not allow unauthorized access to data and resources.

3.**Stress Testing** : Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads

4)**Performance Testing** : Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application.

Product Metrics:

Product metrics are used to evaluate the state of the product, tracing risks and undercovering prospective problem areas. The ability of team to control quality is evaluated

Software Quality:-

Software quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

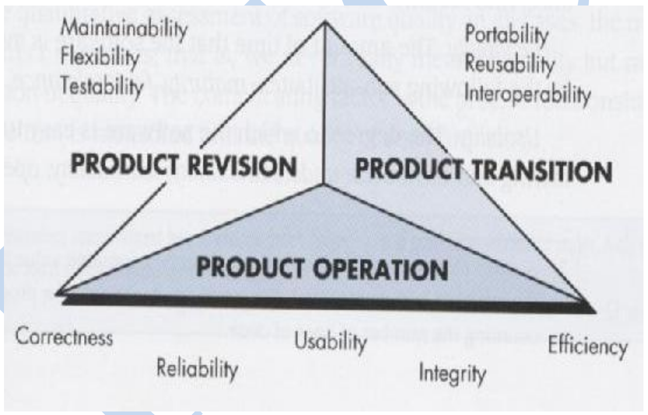
The definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

i). McCall's quality factors

Factors that affect software quality can be categorized in two broad groups:

1. Factors that can be directly measured (e.g. defects uncovered during testing)
2. Factors that can be measured only indirectly (e.g. usability or maintainability)



1. Product operation

i) Correctness:- The extent to which a program satisfies its specification and fulfills the customer's mission objective

ii) Reliability:- The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed

iii) Efficiency:- The amount of computing resources and code required by a program to perform its function.

iv) Integrity:- Extent to which access to software or data by unauthorized persons can be controlled.

v) *Usability*. Effort required to learn, operate, prepare input, and interpret output of a program.

2. Product Revision

i) Maintainability:- Effort required to locate and fix an error in a program. [This is a very limited definition.]

ii) Flexibility:- Effort required to modify an operational program

iii) Testability:- Effort required to test a program to ensure that it performs its intended function.

3. **Product Transition**
   i) *Portability*. Effort required to transfer the program from one hardware and/or software system environment to another.
   ii) *Reusability*. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.
   iii) *Interoperability*. Effort required to couple one system to another.

ii)ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes

1.Functionality
2.Reliability
3.Usability
4.Efficiency
5.Maintainabiliy
6.Portability

1.**Functionality**: It is a key aspect of any product or service. It is due to this the software is able to fulfill a task and keep to its purpose. It is defined as a software product that helps to meet the needs of the clients. A functionality of software is dependent on its complexity. For example: an ATM machine. This is further divided in other categories are as follows:

- Suitability.
- Accuracy.
- Interoperability.
- Security.
- Functional compliance.

2.**Reliability**

Reliability: This characteristic determines the capability of software to sustain its use when put under different circumstances.

3. **Usability**: The usability of software is highly dependent on the functional uses of software. For example: ATM machine is used to withdraw cash. According to the usability of an ATM; the ATM is not affected or influenced by any amounts entered by the user. This is further divided into other sub-categories and these are as follows:

- Maturity.
- Fault Tolerance.
- Recoverability.
- Reliability Compliance.

4. **Efficiency:** This feature of the model is more concerned by resources of the system when used for providing a desired functionality. This type of feature is defined by amount of disk space, memory and network. This is further divided into other sub-categories and these are as follows:

- Understandability.
- Learner ability.

- Operability.
- Attractiveness.
- Usability Compliance.

5. **Maintainability**: This property of maintainability of the software model is used to recognize and fix a defect accordingly. The model is inspected for the faults and these can be identified easily. In accordance to this the cause and effect of maintainability of software is a concern. This is further divided into other sub-categories and these are as follows:

- Analyzability.
- Resource Utilization.
- Stability.
- Testability.
- Changeability.

6. **Portability**: According to this feature, capable software should easily adapt to the environmental changes frequently as possible. The designing of an object and the practices of its implementation are highly dependent on this feature. This standard method is further divided in few categories:

- Adaptability.
- Install ability.
- Co-existence.
- Replaceability.
- Portability compliance.

The full table of Characteristics and Sub characteristics for the ISO 9126-1 Quality Model is:-

Characteristics	**Sub characteristics**	**Definitions**
	Suitability	This is the essential Functionality characteristic and refers to the appropriateness (to specification) of the functions of the software.
	Accurateness	This refers to the correctness of the functions, an ATM may provide a cash dispensing function but is the amount correct?
Functionality	Interoperability	A given software component or system does not typically function in isolation. This subcharacteristic concerns the ability of a software component to interact with other components or systems.
	Compliance	Where appropriate certain industry (or government) laws and guidelines need to be complied with, i.e. SOX. This subcharacteristic addresses the compliant capability of software.
	Security	This subcharacteristic relates to unauthorized access to the software functions.
	Maturity	This subcharacteristic concerns frequency of failure of the software.
Reliability	Fault tolerance	The ability of software to withstand (and recover) from component, or environmental, failure.
	Recoverability	Ability to bring back a failed system to full operation, including data and network connections.
	Understandability	Determines the ease of which the systems functions can be understood, relates to user mental models in Human Computer Interaction methods.

Usability	Learnability	Learning effort for different users, i.e. novice, expert, casual etc.
	Operability	Ability of the software to be easily operated by a given user in a given environment.
Efficiency	Time behavior	Characterizes response times for a given thru put, i.e. transaction rate.
	Resource behavior	Characterizes resources used, i.e. memory, cpu, disk and network usage.
	Analyzability	Characterizes the ability to identify the root cause of a failure within the software.
Maintainability	Changeability	Characterizes the amount of effort to change a system.
	Stability	Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.
	Testability	Characterizes the effort needed to verify (test) a system change.
	Adaptability	Characterizes the ability of the system to change to new specifications or operating environments.
Portability	Installability	Characterizes the effort required to install the software.
	Conformance	Similar to compliance for functionality, but this characteristic relates to portability. One example would be Open SQL conformance which relates to portability of database used.
	Replaceability	Characterizes the *plug and play* aspect of software components, that is how easy is it to exchange a given software component within a specified environment.

Metrics for analysis model:-

Technical work in software engineering begins with the creation of the analysis model. It is at this stage that requirements are derived and that a foundation for design is established. Therefore, technical metrics that provide insight into the quality of the analysis model are desirable.

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics derived for project application for use in this context. These metrics examine the analysis model with the intent of predicting the "size" of the resultant system. It is likely that size and design complexity will be directly correlated.

i)**Function-Based Metrics**

The *function point* (FP) *metric* can be used effectively as a means for measuring the functionality delivered by a system. Using historical data, the FP metric can then be used to

(1) estimate the cost or effort required to design, code, and test the software;
(2) predict the number of errors that will be encountered during testing; and
(3) forecast the number of components and/or the number of projected source lines in the implemented system.

Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity. Information domain values are defined in the following manner

Number of external inputs (EIs). Each *external input* originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update internal logical files (ILFs). Inputs should be distinguished from inquiries, which are counted separately.
Number of external outputs (EOs). Each *external output* is derived data within the application that provides information to the user. In this context external output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.
Number of external inquiries (EQs). An *external inquiry* is defined as an online input that results in the generation of some immediate software response in the form of an online output (often retrieved from an ILF).
Number of internal logical files (ILFs). Each *internal logical file* is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

Number of external interface files (EIFs). Each *external interface file* is a logical grouping of data that resides external to the application but provides information that may be of use to the application.

Once these data have been collected, the table in Figure 23.1 is completed and a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective. To compute function points (FP), the following relationship is used:

$$\text{FP} = \text{count total}\ [0.65 + 0.01 * \sum (F_i)]$$

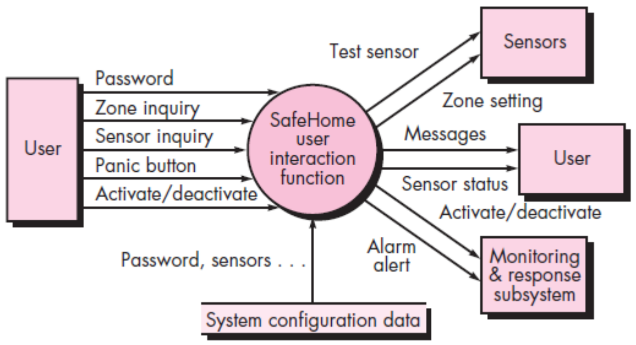
Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total ⟶							

The *Fi* (*i* _ 1 to 14) are *value adjustment factors* (VAF) based on responses to the following questions [Lon02]:

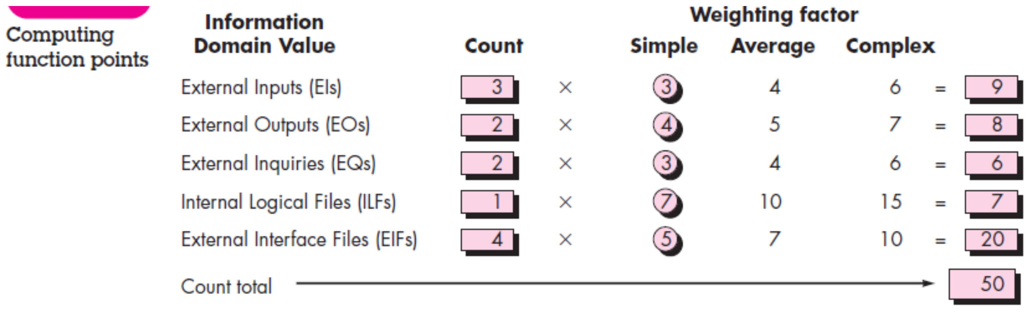
1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in figure. Referring to the figure, a data flow diagram for a function within the SafeHome software is represented. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The function displays a series of prompting messages and sends appropriate control signals to various components of the security system.



A data flow model for *SafeHome* software

The data flow diagram is evaluated to determine a set of key information domain measures required for computation of the function point metric. Three external inputs—**password, panic button**, and **activate/deactivate**—are shown in the figure along with two external inquiries—**zone inquiry** and **sensor inquiry**. One ILF (**system configuration file**) is shown. Two external outputs (**messages** and **sensor status**) and four EIFs (**test sensor, zone setting, activate/deactivate,** and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in below Figure


Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)	3	×	[✓ 3]	4	6	=	9
External Outputs (EOs)	2	×	[✓ 4]	5	7	=	8
External Inquiries (EQs)	2	×	[✓ 3]	4	6	=	6
Internal Logical Files (ILFs)	1	×	[✓ 7]	10	15	=	7
External Interface Files (EIFs)	4	×	[✓ 5]	7	10	=	20
Count total						→	50

The count total shown in Figure 23.3 must be adjusted using Equation (23.1). For the purposes of this example, we assume that _(*Fi*) is 46 (a moderately complex product). Therefore,

$$FP = 50\ [0.65 + (0.01\ 46)] = 56$$

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the SafeHome user interaction function. Assume that past data indicates that one FP translates into 60 lines of code (an objectoriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the analysis model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during analysis and design reviews and four errors per function point during unit and integration testing. These data can help software engineers assess the completeness of their review and testing activities.

ii)**Metrics for Specification Quality**

Davis and his colleagues propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: specificity (lack of ambiguity), completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability. In addition, the authors note that high-quality specifications are electronically stored, executable or at least interpretable, annotated by relative importance and stable, versioned, organized, cross-referenced, and specified at the right level of detail.

Although many of these characteristics appear to be qualitative in nature, Davis et al. suggest that each can be represented using one or more metrics. For example, we assume that there are nr requirements in a specification, such that

$$n_r = n_f + n_{nf}$$

where n_f is the number of functional requirements and n_{nf} is the number of nonfunctional (e.g., performance) requirements.

To determine the specificity (lack of ambiguity) of requirements, Davis et al. suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$$Q_1 = n_{ui}/n_r$$

where n_{ui} is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

The completeness of functional requirements can be determined by computing the ratio

$$Q_2 = n_u/[n_i \text{ x } n_s]$$

where nu is the number of unique function requirements, n_i is the number of inputs (stimuli) defined or implied by the specification, and ns is the number of states specified. The Q2 ratio measures the percentage of necessary functions that have been specified for a system. However, it does not address nonfunctional requirements. To incorporate these into an overall metric for completeness, we must consider the degree to which requirements have been validated:

$Q_3 = n_c/[n_c + n_{nv}]$

where nc is the number of requirements that have been validated as correct and nnv is the number of requirements that have not yet been validated.

Metrics for Design Model:-

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative .

We can examine some of the more common design metrics for computer software. Each can provide the designer with improved insight and all can help the design to evolve to a higher level of quality.

i)Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are “black box” in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass define three software design complexity measures: structural complexity, data complexity, and system complexity.

Structural complexity of a module i is defined in the following manner:

S(i) = f 2 out(i)

where fout(i) is the fan-out7 of module i.

Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

D(i) = v(i)/[fout(i) +1]

where v(i) is the number of input and output variables that are passed to and from module i.
Finally, **system complexity** is defined as the sum of structural and data complexity, specified as

C(i) = S(i) + D(i)

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

An earlier high-level architectural design metric proposed by Henry and Kafura also makes use the fan-in and fan-out. The authors define a complexity metric (applicable to call and return architectures) of the form

HKM = length(i) x [fin(i) + fout(i)]2

where length(i) is the number of programming language statements in a module i and fin(i) is the fan-in of a module i. Henry and Kafura extend the definitions of fanin and fan-out presented in this book to include not only the number of module control connections (module calls) but also the number of data structures from which a module i retrieves (fan-in) or updates (fan-out) data. To compute HKM during design, the procedural design may be used to estimate the number of programming language statements for module i. Like the Card and Glass metrics noted previously, an increase in the Henry-Kafura metric leads to a greater likelihood that integration and testing effort will also increase for a module.

Fenton suggests a number of simple morphology (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to figure, the following metrics can be defined:

size = n + a

where n is the number of nodes and a is the number of arcs. For the architecture shown in figure,

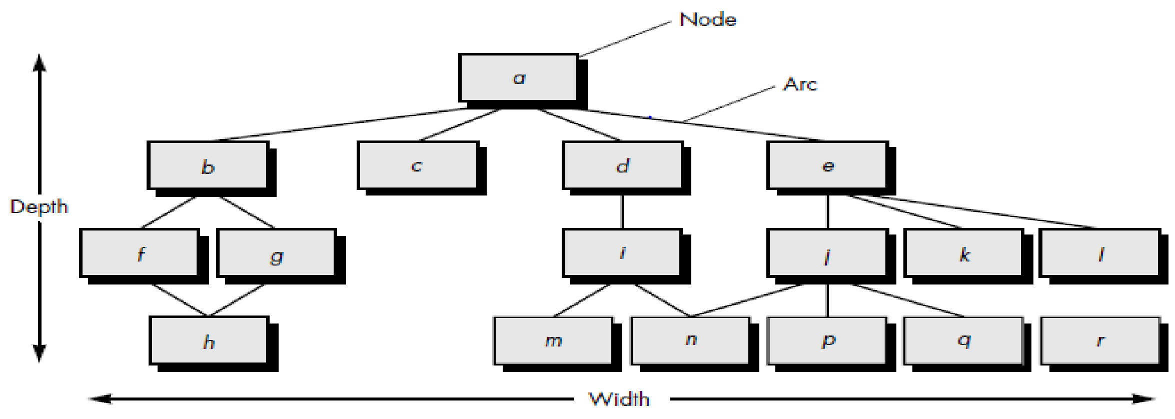


FIGURE 19.5 Morphology metrics

size = 17 + 18 = 35

depth = the longest path from the root (top) node to a leaf node. For the architecture shown in figure, depth = 4.

width = maximum number of nodes at any one level of the architecture. For the architecture shown in figure, width = 6.

arc-to-node ratio, r = a/n,

which measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture. For the architecture shown in figure, r = 18/17 = 1.06.

The U.S. Air Force Systems Command] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. Using concepts similar to those proposed in IEEE Std. 982.1-1988 , the Air Force uses information obtained from data and architectural design to derive a design structure quality index (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI :

S1 = the total number of modules defined in the program architecture.
S2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S2).
S3 = the number of modules whose correct function depends on prior processing.
S4 = the number of database items (includes data objects and all attributes that define objects).
S5 = the total number of unique database items.
S6 = the number of database segments (different records or individual objects).
S7 = the number of modules with a single entry and exit (exception processing is not considered to be a multiple exit).

Once values S1 through S7 are determined for a computer program, the following intermediate values can be computed:

Program structure: D1, where D1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then D1 = 1, otherwise D1 = 0.

Module independence: D2 = 1 (S2/S1)

Modules not dependent on prior processing: D3 = 1 (S3/S1)

Database size: D4 = 1 (S5/S4)

Database compartmentalization: D5 = 1 (S6/S4)

Module entrance/exit characteristic: D6 = 1 (S7/S1)

With these intermediate values determined, the DSQI is computed in the following manner:

DSQI = wiDi

where i = 1 to 6, wi is the relative weighting of the importance of each of the intermediate values, and wi = 1 (if all Di are weighted equally, then wi = 0.167).

The value of DSQI for past designs can be determined and compared to a design that is currently under development. If the DSQI is significantly lower than average, further design work and review are indicated. Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

ii)Component-Level Design Metrics

Component-level design metrics focus on internal characteristics of a software component and include measures of the "three Cs"—module cohesion, coupling, and complexity. These measures can help a software engineer to judge the quality of a component-level design.

The metrics presented in this section are glass box in the sense that they require knowledge of the inner working of the module under consideration. Component-level design metrics may be applied once a procedural design has been developed. Alternatively, they may be delayed until source code is available.

Cohesion metrics.

Bieman and Ott define a collection of metrics that provide an indication of the cohesiveness of a module. The metrics are defined in terms of five concepts and measures:

a. Data slice. Stated simply, a data slice is a backward walk through a module that looks for data values that affect the module location at which the walk began. It should be noted that both program slices (which focus on statements and conditions) and data slices can be defined.

b. Data tokens. The variables defined for a module can be defined as data tokens for the module.

c. Glue tokens. This set of data tokens lies on one or more data slice.
d. Superglue tokens. These data tokens are common to every data slice in a module.
e. **Stickiness**. The relative stickiness of a glue token is directly proportional to the number of data slices that it binds.

Bieman and Ott develop metrics for strong functional cohesion (SFC), weak functional cohesion (WFC), and adhesiveness (the relative degree to which glue tokens bind data slices together). These metrics can be interpreted in the following manner :

All of these cohesion metrics range in value between 0 and 1. They have a value of 0 when a procedure has more than one output and exhibits none of the cohesion attribute indicated by a particular metric. A procedure with no superglue tokens, no tokens that are common to all data slices, has zero strong functional cohesion—there are no data tokens that contribute to all outputs. A procedure with no glue tokens, that is no tokens common to more than one data slice (in procedures with more than one data slice), exhibits zero weak functional cohesion and zero adhesiveness—there are no data tokens that contribute to more than one output.

Strong functional cohesion and adhesiveness are encountered when the Bieman and Ott metrics take on a maximum value of 1.

To illustrate the character of these metrics, consider the metric for strong functional cohesion:

SFC(i) = SG [SA(i))/(tokens(i)]

where SG[SA(i)] denotes superglue tokens—the set of data tokens that lie on all data slices for a module i. As the ratio of superglue tokens to the total number of tokens in a module i increases toward a maximum value of 1, the functional cohesiveness of the module also increases.

Coupling metrics.

Module coupling provides an indication of the "connectedness" of a module to other modules, global data, and the outside environment. Coupling was discussed in qualitative terms.

Dhama has proposed a metric for module coupling that encompasses data and control flow coupling, global coupling, and environmental coupling. The measures required to compute module coupling are defined in terms of each of the three coupling types noted previously.

For data and control flow coupling,
d_i = number of input data parameters
c_i = number of input control parameters
d_0 = number of output data parameters
c_0 = number of output control parameters

For global coupling,
g_d = number of global variables used as data
g_c = number of global variables used as control

For environmental coupling,
w = number of modules called (fan-out)
r = number of modules calling the module under consideration (fan-in)

Using these measures, a module coupling indicator, mc , is defined in the following way:

$$mc = k/M$$

where k = 1, a proportionality constant and

$$M = d_i + (a \times c_i) + d_0 + (b \times c_0) + g_d + (c \times g_c) + w + r$$

where a = b = c = 2.

The higher the value of mc, the lower is the overall module coupling. For example, if a module has single input and output data parameters, accesses no global data, and is called by a single module,

$$mc = 1/(1 + 0 + 1+ 0 + 0 + + 0 + 1 + 0) = 1/3 = 0.33$$

We would expect that such a module exhibits low coupling. Hence, a value of mc = 0.33 implies low coupling. Alternatively, if a module has five input and five output data parameters, an equal number of control parameters, accesses ten items of global data, has a fan-in of 3 and a fan-out of 4,

$$mc = 1/[5 + (2\ 5) + 5 + (2\ 5) + 10 + 0 + 3 + 4] = 0.02$$

and the implied coupling would be high.

In order to have the coupling metric move upward as the degree of coupling increases , a revised coupling metric may be defined as

$$C = 1 - m_c$$

where the degree of coupling increases nonlinearly between a minimum value in the range 0.66 to a maximum value that approaches 1.0.

Complexity metrics.

A variety of software metrics can be computed to determine the complexity of program control flow. Many of these are based on the flow graph. A graph is a representation composed of nodes and links (also called edges). When the links (edges) are directed, the flow graph is a directed graph.

McCabe and Watson identify a number of important uses for complexity metrics:

Complexity metrics can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of source code [or procedural design information]. Complexity metrics also provide feedback during the software project to help control the [design activity]. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability.

The most widely used (and debated) complexity metric for computer software is cyclomatic complexity, originally developed by Thomas McCabe .

The McCabe metric provides a quantitative measure of testing difficulty and an indication of ultimate reliability. Experimental studies indicate distinct relationships between the McCabe metric and the number of errors existing in source code, as well as time required to find and correct such errors.

McCabe also contends that cyclomatic complexity may be used to provide a quantitative indication of maximum module size. Collecting data from a number of actual programming projects, he has found that cyclomatic complexity = 10 appears to be a practical upper limit for module size. When the cyclomatic complexity of modules exceeded this number, it became extremely difficult to adequately test a module.

Zuse presents an encyclopedic discussion of no fewer that 18 different categories of software complexity metrics. The author presents the basic definitions for metrics in each category (e.g., there are a number of variations on the cyclomatic complexity metric) and then analyzes and critiques each. Zuse's work is the most comprehensive published to date.

iii)Interface Design Metrics:-

There are significant literature on the design of human/computer interfaces relatively little information has been published on metrics that would provide insight into the quality and usability of the interface.

layout appropriateness (LA) is a worthwhile design metric for human/computer interfaces. A typical GUI uses *layout entities*—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the "cost" of the transition from one layout entity to the next all contribute to the appropriateness of the interface.

For a specific layout (i.e., a specific GUI design), cost can be assigned to each sequence of actions according to the following relationship:

cost = $\sum$ [frequency of transition(k) × cost of transition(k)]

where k is a specific transition from one layout entity to the next as a specific task is accomplished. The summation occurs across all transitions for a particular task or set of tasks required to accomplish some application function. Cost may be characterized in terms of time, processing delay, or any other reasonable value, such as the distance that a mouse must travel between layout entities. Layout appropriateness is defined as

LA = 100 × [(cost of LA _ optimal layout)/(cost of proposed layout)]

where LA = 100 for an optimal layout.

To compute the optimal layout for a GUI, interface real estate (the area of the screen) is divided into a grid. Each square of the grid represents a possible position for a layout entity. For a grid with N possible positions and K different layout entities to place, the number of possible layouts is represented in the following manner [SEA93]:

number of possible layouts = $[N!/(K! * (N - K)!] * K!$

As the number of layout positions increases, the number of possible layouts grows very large. To find the optimal (lowest cost) layout, Sears [SEA93] proposes a tree searching algorithm.

LA is used to assess different proposed GUI layouts and the sensitivity of a particular layout to changes in task descriptions (i.e., changes in the sequence and/or frequency of transitions). The interface designer can use the change in layout appropriateness, ΔLA, as a guide in choosing the best GUI layout for a particular application.

It is important to note that the selection of a GUI design can be guided with metrics such as LA, but the final arbiter should be user input based on GUI prototypes. Nielsen and Levy [NIE94] report that "one has a reasonably large chance of success if one chooses between interface [designs] based solely on users' opinions. Users' average task performance and their subjective satisfaction with a GUI are highly correlated."

METRIC FOR SOURCE CODE:-

Halstead assigned quantitative laws to the development of computer software using a set of primitive's measures which can be derived after code is generated or estimated once design is complete

The primitive measures are

n1 = the number of distinct operators that appear in a program
n2 = the number of distinct operands that appear in a program
N1 = the total number of operator occurrences.
N2 = the total number of operand occurrence.

These primitives are used to develop expression for the

=> Overall program length
=> Potential minimum volume for an algorithm
=> the actual volume [number bits required to specify a program]
=> Program level [a measure of software complexity]
=> Development effort
=> Development time etc..

Halsted shows that Length N can be estimated

$N = n1 \log_2 n1 + n2 \log_2 n2$

The program volume may be defined by

$V = N \log2 (n1 + n2)$

Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

$L = 2/n_1 \times n_2/N_2$

METRIC FOR TESTING:-

Whenever we consider the metrics for testing, it as to be noted that it is derived by considering the implementation of process of testing.

i). Halstead metrics applied to testing:

Using the definitions for program volume V and program level PL, Halstead effort e can be computed as

Program Level and Effort

$PL = 1/[(n1 / 2) \times (N2 / n2)]$

$e = V/PL$

The overall testing effort to be allocated to a module K, can be estimated using

Percentage of testing effect (K) = e(K) / ∑e(i)

Where,

=> e(k) is computed for module K using PL

=> Summation in denominator is the sum of halstead effort across all modules of the system.

ii).Object Oriented Testing Metrics:

Testing metrics can be grouped into two categories: encapsulation and inheritance. Encapsulation

Lack of cohesion in methods (LCOM) **-** The higher the value of LCOM, the more states have to be tested.

Percent public and protected (PAP) **-** This number indicates the percentage of class attributes that are public and thus the likelihood of side effects among classes.

Public access to data members (PAD) **-** This metric shows the number of classes that access other class's attributes and thus violation of encapsulation Inheritance

Number of root classes (NOR) **-** A count of distinct class hierarchies.

Fan in (FIN) **-** FIN > 1 is an indication of multiple inheritance and should be avoided.

Number of children (NOC) and depth of the inheritance tree (DIT) - For each subclass, its superclass has to be re-tested. The above metrics (and others) are different than those used in traditional software testing, however, metrics collected from testing should be the same (i.e. number and type of errors, performance metrics, etc.).

METRICS FOR MAINTENANCE:-

IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

M_t = the number of modules in the current release

F_c = the number of modules in the current release that have been changed

F_a = the number of modules in the current release that have been added.

F_d = the number of modules from the preceding release that were deleted in the current release

The Software Maturity Index, SMI, is defined as:

$$SMI = [M_t - (F_a + F_c + F_d)] / M_t$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

METRICS FOR PROCESS AND PROJECTS

- **Process metrics** − These characteristics can be used to improve the development and maintenance activities of the software.
- **Project metrics** − This metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

SOFTWARE MEASUREMENT:-

Software measurement can be categorized in two ways.

(1) ***Direct measures*** of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.

(2) ***Indirect measures*** of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities"

i)Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced.
To develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Project	**LOC**	**Effort**	**$ (000)**	**PP.doc**	**Errors**	**Defects**	**People**
Alpha	12100	24	168	365	134	29	3
Beta	27200	62	440	1224	321	86	5
Gamma	20200	43	314	1050	256	64	6
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

ii)Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the ***function point***. **Function points** are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

- ✓ **Proponents claim** that FP is programming language independent, making it ideal for application using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.
- ✓ **Opponents claim** that the method requires some "sleight of hand " in that computation is based subjective rather than objective data, that counts of the information domain can be difficult to collect after the fact, and that FP has no direct physical meaning- it's just a number.

Typical Function-Oriented Metrics:

- errors per FP (thousand lines of code)
- defects per FP
- $ per FP
- pages of documentation per FP
- FP per person-month

iii)Reconciling Different Metrics Approaches

The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost.

Programming Language	**LOC per Function Point**			
	AVG	**MEDIAN**	**LOW**	**HIGH**
Access	35	38	15	47
Ada	154	-	104	200
Asp	650	315	91	64
.	.	.	.	.
.	.	.	.	.

iv)Object Oriented Metrics:

Conventional software project metrics (LOC or FP) can be used to estimate object oriented software projects. Lorenz and Kidd suggest the following set of metrics for OO projects:

- **Number of scenario scripts**: A scenario script is a detailed sequence of steps that describes the interaction between the user and the application.
- **Number of key classes**: Key classes are the "highly independent components that are defined early in object-oriented analysis.
- **Number of support classes**: Support classes are required to implement the system but are not immediately related to the problem domain.
- **Average number of support classes per key class**: Of the average number of support classes per key class were known for a given problem domain estimation would be much simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes.
- **Number of subsystems**: A subsystem is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

v)Use-Case Oriented Metrics

Use-cases describe user-visible functions and features that are basic requirements for a system. The use-cases is directly proportional to the size of the application in LOC and to the number of use-cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use-cases can be created at vastly different levels of abstraction, there is no standard size for a use-case. Without a standard measure of what a use-case is, its application as a normalization measure is suspect.

vi)Web Engineering Project Metrics

The objective of all web engineering projects is to build a Web application that delivers a combination of content and functionality to the end-user.

- **Number of static Web pages**: These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measures provides an indication of the overall size of the application and the effort required to develop it.
- **Number of dynamic Web pages**: : Web pages with dynamic content are essential in all e-commerce applications, search engines, financial application, and many other Web App categories. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.
- **Number of internal page link**: Internal page links are pointers that provide an indication of the degree of architectural coupling within the Web App.
- **Number of persistent data objects**: As the number of persistent data objects grows, the complexity of the Web App also grows, and effort to implement it increases proportionally.
- **Number of external systems interfaced**: As the requirement for interfacing grows, system complexity and development effort also increase.
- **Number of static content objects**: Static content objects encompass static text- based, graphical, video, animation, and audio information that are incorporated within the Web App.
- **Number of dynamic content objects**: Dynamic content objects are generated based on end-user actions and encompass internally generated text-based, graphical, video, animation, and audio information that are incorporated within the Web App.
- **Number of executable functions**: An executable function provides some computational service to the end-user. As the number of executable functions increases, modeling and construction effort also increase.

Each of the preceding measures can be determined at a relatively early stage. For example, you can define a metric that reflects the degree of end-user customization that is required for the WebApp and correlate it to the effort expended on the project and/or the errors uncovered as reviews and testing are conducted. To accomplish this, you define

N_{sp} = number of Static Web pages

N_{dp} = number of Dynamic Web pages

Then,

Customization index, $C = N_{dp}/(N_{dp}+N_{sp})$

The value of C ranges from 0 to 1. As C grows larger, the level of WebApp customization becomes a significant technical issue

METRICS FOR SOFTWARE QUALITY:-

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process.

i) Measuring Quality

The measures of software quality are correctness, maintainability, integrity, and usability. These measures will provide useful indicators for the project team.

- **Correctness.** Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- **Maintainability.** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in

requirements. A simple time-oriented metric is ***mean-time-tochange*** (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.

- **Integrity.** Attacks can be made on all three components of software: programs, data, and documents.

  To measure integrity, two additional attributes must be defined: **threat** and **security.**

  Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time.

  Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

  $$\text{Integrity} = \sum [1 - (\text{threat} \times (1 - \text{security}))]$$

- **Usability:** Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

ii) Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$\text{DRE} = E/(E + D)$$

where E is the number of errors found before delivery of the software to the end-user and
D is the number of defects found after delivery.

Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

$$\text{DRE}_i = E_i/(E_i+E_i+1)$$

E_i is the number of errors found during software engineering activity *i* and
E_{i+1} is the number of errors found during software engineering activity *i+1* that are traceable to errors that were not discovered in software engineering activity *i*.

A quality objective for a software team (or an individual software engineer) is to achieve DRE that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

Module - V

RISK MANAGEMENT

Management of Risk Process:

REACTIVE VS. PROACTIVE RISK STRATEGIES:-

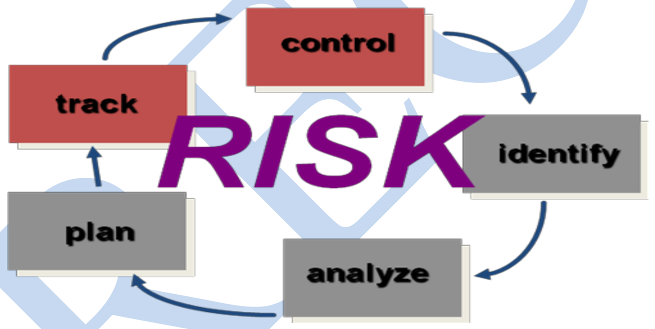
At best, a **reactive strategy** monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire fighting mode.*

- project team reacts to risks when they occur
- mitigation—plan for additional resources in anticipation of fire fighting
- fix on failure—resource are found and applied when the risk strikes
- crisis management—failure does not respond to applied resources and project is in jeopardy

A **proactive strategy** begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk.

- formal risk analysis is performed
- organization corrects the root causes of risk
    - examining risk sources that lie beyond the bounds of the software
    - developing the skill to manage change

Risk Management Paradigm



SOFTWARE RISK:-

Risk always involves two characteristics
Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks
Loss—if the risk becomes a reality, unwanted consequences or losses will occur.
When risks are analyzed, it is important to quantify the level of uncertainty in the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.
Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase.
Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.
Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are
(1) Building a excellent product or system that no one really wants (market risk),
(2) Building a product that no longer fits into the overall business strategy for the company (strategic risk),
(3) Building a product that the sales force doesn't understand how to sell,
(4) Losing the support of senior management due to a change in focus or a change in people (management risk), and

(5) Losing budgetary or personnel commitment (budget risks).

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources.

Predictable risks are extrapolated from past project experience.

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

RISK IDENTIFICATION:-

Risk identification is a systematic attempt to specify threats to the project plan. There are two distinct types of risks.

1) Generic risks and
2) product-specific risks.

Generic risks are a potential threat to every software project.
Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project that is to be built.
Known and predictable risks in the following generic subcategories:

- ***Product size***—risks associated with the overall size of the software to be built or modified.
- ***Business impact***—risks associated with constraints imposed by management or the marketplace.
- ***Customer characteristics***—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- ***Process definition***—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.
- ***Technology to be built***—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- ***Staff size and experience***—risks associated with the overall technical and project experience of the software engineers who will do the work.

i).Assessing Overall Project Risk

The questions are ordered by their relate importance to the success of a project.
1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and their customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be
Implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

ii). Risk Components and Drivers

The risk components are defined in the following manner:
• ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
• ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
• ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
• ***Schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

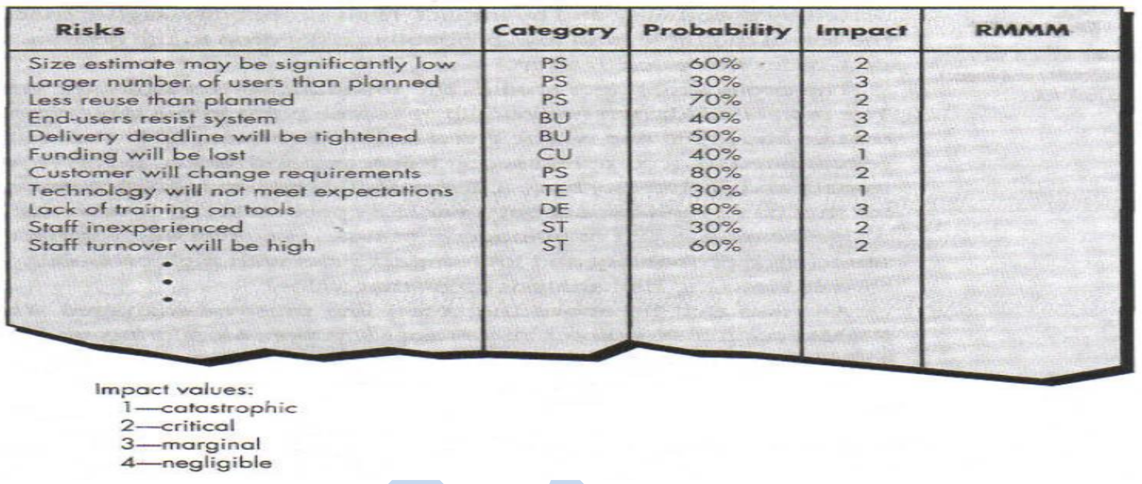
The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic.

RISK PROJECTION:-

Risk projection, also called ***risk estimation***, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur.
The project planner, along with other managers and technical staff, performs four risk projection activities:
(1) establish a scale that reflects the perceived likelihood of a risk,
(2) delineate the consequences of the risk,
(3) estimate the impact of the risk on the project and the product, and
(4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

i).Developing a Risk Table

Risk table provides a project manager with a simple technique of risk projection.
A simple risk table is illustrated in figure below,



Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
• • •				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Figure , Sample Risk Table prior to sorting

- A project team begins by listing all risks (no matter how remote) in the first column of the table.
- Each risk is categorized in Next; the impact of each risk is assessed.
- The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.
- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

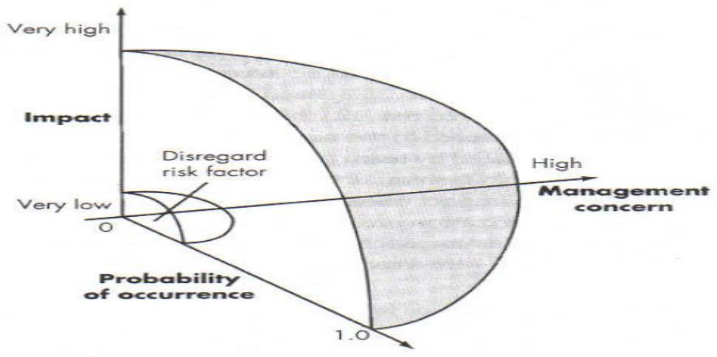


Figure , Risk and Management concern

The project manager studies the resultant sorted table and defines a cutoff line.
The ***cutoff line*** (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization.

ii).Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing.

- ✓ The ***nature*** of the risk indicates the problems that are likely if it occurs.
- ✓ The ***scope*** of a risk combines the severity (just how serious is it?) with its overall distribution.
- ✓ Finally, the ***timing*** of a risk considers when and for how long the impact will be felt.

The overall ***risk exposure***, RE, is determined using the following relationship
RE = P x C
Where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80% (likely).

Risk impact. 60 reusable software components were planned.

Risk exposure. *RE* = 0.80 x 25,200 ~ $20,200.

The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project etc.

RISK REFINEMENT:-

One way for risk refinement is to represent the risk in *condition-transition-consequence(CTC)* format.
This general condition can be refined in the following manner:
Sub condition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
Sub condition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Sub condition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

RISK MITIGATION, MONITORING, AND MANAGEMENT:-

An effective strategy must consider three issues:

- Risk avoidance
- Risk monitoring
- Risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy.
To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under our control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed"). • Assign a backup staff member for every critical technologist.

As the project proceeds, risk monitoring activities commence. The following factors can be monitored:

- General attitude of team members based on project pressures.
- The degree to which the team has jelled.
- Interpersonal relationships among team members.

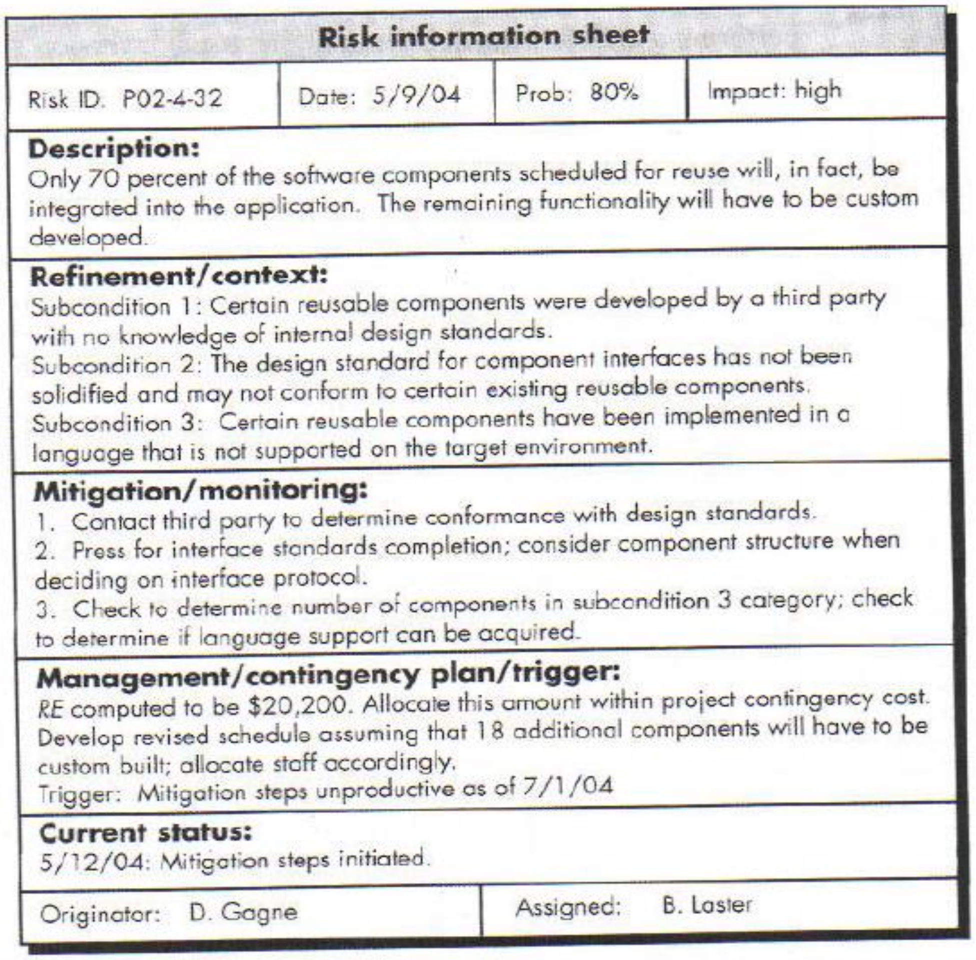
- Potential problems with compensation and benefits
- The availability of jobs within the company and outside it.

Software safety **and** ***hazard analysis*** are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

THE RMMM PLAN:-

A risk management strategy can be included in the software project plan or the risk management steps can be organized into a separate *Risk Mitigation, Monitoring and Management Plan.*

The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.



Risk information sheet

Risk ID: P02-4-32	Date: 5/9/04	Prob: 80%	Impact: high

Description:
Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Refinement/context:
Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards.
Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.

Mitigation/monitoring:
1. Contact third party to determine conformance with design standards.
2. Press for interface standards completion; consider component structure when deciding on interface protocol.
3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.

Management/contingency plan/trigger:
RE computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly.
Trigger: Mitigation steps unproductive as of 7/1/04

Current status:
5/12/04: Mitigation steps initiated.

Originator: D. Gagne	Assigned: B. Laster

Figure , Risk information Sheet

Risk monitoring is a project tracking activity with three primary objectives:
1) to assess whether predicted risks do, in fact, occur;
2) to ensure that risk aversion steps defined for the risk are being properly applied; and
3) to collect information that can be used for future risk analysis.

QUALITY MANAGEMENT

QUALITY CONCEPTS:-

Quality management encompasses
(1) a quality management approach,
(2) effective software engineering technology (methods and tools),
(3) formal technical reviews that are applied throughout the software process,
(4) a multitiered testing strategy,
(5) control of software documentation and the changes made to it,
(6) a procedure to ensure compliance with software development standards (when applicable), and
(7) measurement and reporting mechanisms.

Variation control is the heart of quality control.

i). Quality

The *American Heritage Dictionary* defines *quality* as "a characteristic or attribute of something."

- ***Quality of design*** refers to the characteristics that designers specify for an item.
- ***Quality of conformance*** is the degree to which the design specifications are followed during manufacturing.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.
Robert Glass argues that a more "intuitive" relationship is in order:

User satisfaction = compliant product + good quality + delivery within budget and schedule

ii). Quality Control

Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.
A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

iii). Quality Assurance

Quality assurance consists of the auditing and reporting functions that assess the effectiveness and completeness of quality control activities. The **goal of quality** assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.

iv). Cost of Quality

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities.

Quality costs may be divided into costs associated with prevention, appraisal, and failure.
Prevention costs include

- quality planning
- formal technical reviews
- test equipment
- training

Appraisal costs include activities to gain insight into product condition the “first time through” each process. Examples of appraisal costs include

- in-process and interprocess inspection
- equipment calibration and maintenance
- testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs.

Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework
- repair
- failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are

- complaint resolution
- product return and replacement
- help line support
- warranty work

SOFTWARE QUALITY ASSURANCE:-

Software quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

The definition serves to emphasize three important points:

1) Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2) Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surelyresult.
3) A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

i). Background Issues

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.Today, every company has mechanisms to ensure quality in its products.

During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s.

Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view

ii). SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—

- the software engineers who do technical work and
- an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that conducts the following activities.

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

- ✓ evaluations to be performed
- ✓ audits and reviews to be performed
- ✓ standards that are applicable to the project
- ✓ procedures for error reporting and tracking
- ✓ documents to be produced by the SQA group
- ✓ amount of feedback provided to the software project team

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

SOFTWARE REVIEWS:-

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called *analysis*, *design,* and *coding.*

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review
A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

i). Cost Impact of Software Defects:

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.

A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors during the software process. However, formal review techniques have been shown to be up to 75 percent effective] in uncovering design errors. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and support phases.

To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects Assume that an error uncovered

- ✓ during design will cost 1.0 monetary unit to correct.
- ✓ just before testing commences will cost 6.5 units;

- ✓ during testing, 15 units;
- ✓ and after release, between 60 and 100 units.

ii). Defect Amplification and Removal:

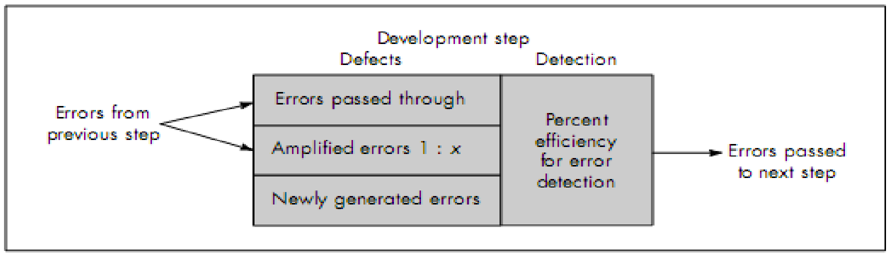


Figure , Defect Amplification Model

A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process.

A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

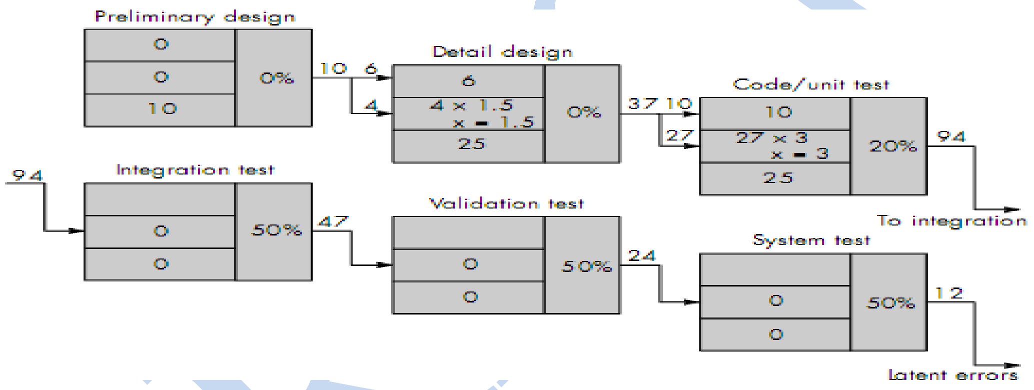


Figure A , Defect Amplification, no reviews

Referring to the figure A, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field.

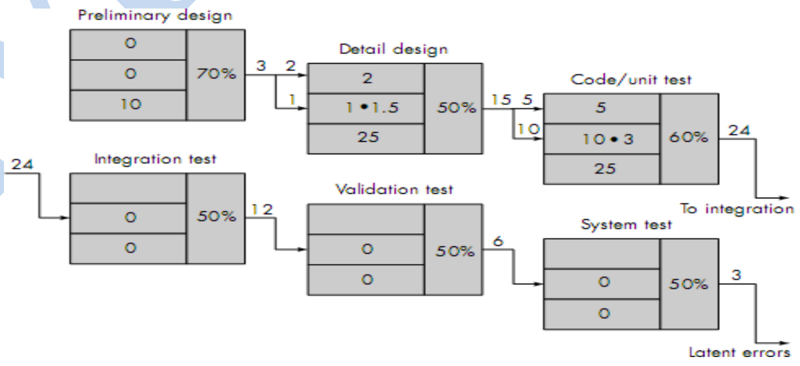


Figure B , Defect amplification, reviews conducted

Figure B, considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent errors exist.

Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 8.3 and 8.4 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release).

- ✓ ***Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units.***
- ✓ ***When no reviews are conducted, total cost is 2177 units—nearly three times more costly.***

To conduct reviews, a software engineer must expend time and effort and the development organization must spend money. Formal technical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

FORMAL TECHNICAL REVIEWS:-

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are
(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verifythat the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner; and
(5) to make projects more manageable.

i). The Review Meeting

Every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

The focus of the FTR is on a work product.
The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required.

- ✓ The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
- ✓ Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- ✓ The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review.

At the end of the review, all attendees of the FTR must decide whether to
(1) accept the product without further modification,
(2) reject the product due to severe errors (once corrected, another review must be performed), or
(3) accept the product provisionally.
The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

ii). Review Reporting and Record Keeping

At the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A ***review summary report*** answers three questions:
1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?
The review summary report is a single page form.
It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

iii). Review Guidelines

The following represents a minimum set of guidelines for formal technical reviews:

1. ***Review the product, not the producer***. An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment.
2. ***Set an agenda and maintain it***. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. ***Limit debate and rebuttal***. When an issue is raised by a reviewer, there may not be universal agreement on its impact.
4. ***Enunciate problem areas, but don't attempt to solve every problem noted***. A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. ***Take written notes***. It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. ***Limit the number of participants and insist upon advance preparation***. Keep the number of people involved to the necessary minimum.
7. ***Develop a checklist for each product that is likely to be reviewed***. A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even test documents.
8. ***Allocate resources and schedule time for FTRs***. For reviews to be effective, they should be scheduled as a task during the software engineering process
9. ***Conduct meaningful training for all reviewers.*** To be effective all review participants should receive some formal training.
10. ***Review your early reviews***. Debriefing can be beneficial in uncovering problems with the review process itself.

iv). Sample-Driven Reviews (SDRs):

SDRs attempt to quantify those work products that are primary targets for full FTRs. To accomplish this the following steps are suggested…

- Inspect a fraction a_i of each software work product, *i*. Record the number of faults, f_i found within a_i.
- Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by$1/a_i$.
- Sort the work products in descending order according to the gross estimate of the number of faults in each.
- Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must

- Be representative of the work product as a whole and
- Large enough to be meaningful to the reviewer(s) who does the sampling.

STATISTICAL SOFTWARE QUALITY ASSURANCE:-

For software, statistical quality assurance implies the following steps:

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the defects

This relatively simple concept represents an important step towards the creation o: an adaptive software engineering process in which changes are made to improve those elements of the process that introduce error.

i). A Generic Example:

To illustrate this, assume that a software engineering organization collects information on defects for a period of one year. Some of the defects are uncovered as software is being developed. Others are encountered after the software has been released to its end-users.

Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

•incomplete or erroneous specifications (IBS)
•misinterpretation of customer communication (MCC)
•intentional deviation from specifications (IDS)
•violation of programming standards (VPS)
•error in data representation (EDR)
•inconsistent component interface (ICI)
•error in design logic (EDL)
•incomplete or erroneous testing (IET)
•inaccurate or incomplete documentation (IID)
•error in programming language translation of design (PLT)
•ambiguous or inconsistent human/computer interface (HCI)
•miscellaneous (MIS)

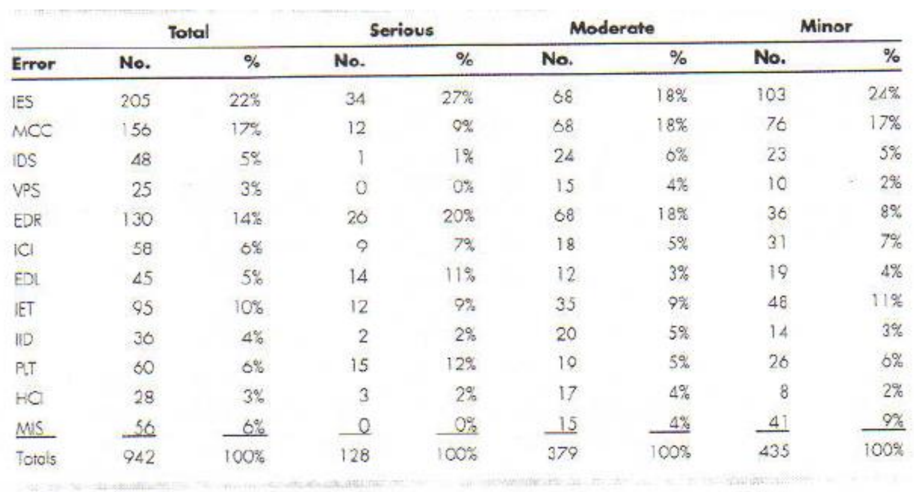
To apply statistical SQA, Table 1.1 is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action.

For example, to correct MCC, the software developer might implement facilitated application specification techniques to improve the quality of customer communication and specifications.

To improve EDR, the developer might acquire CASE tools for data modeling and perform more stringent data design reviews. It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement [ART97]. In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

Table 1.1: DATA COLLECTION FOR STATISTICAL SQA



	Total		Serious		Moderate		Minor	
Error	No.	%	No.	%	No.	%	No.	%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

The application of the statistical SQA and the pareto principle can be summarized in a single sentence: *spend your time focusing on things that really matter, but first be sure that you understand what really matters*.

ii). Six Sigma for software Engineering:

Six Sigma is the most widely used strategy for statistical quality assurance in industry today.
The term "six sigma" is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

- ***Define*** customer requirements and deliverables and project goals via well-defined methods of customer communication
- ***Measure*** the existing process and its output to determine current quality performance (collect defect metrics)
- ***Analyze*** defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps.

- ***Improve*** the process by eliminating the root causes of defects.
- ***Control*** the process to ensure that future work does not reintroduce the causes of defects

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If any organization is developing a software process (rather than improving and existing process), the core steps are augmented as follows:

- ***Design*** the process to
  - avoid the root causes of defects and
  - to meet customer requirements
- ***Verify*** that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design and verify)method.

SOFTWARE RELIABILITY:-

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

i). Measures of Reliability and Availability:

Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.
A simple measure of reliability is *meantime-between-failure* (MTBF), where

MTBF = MTTF + MTTR

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

In addition to a reliability measure, we must develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

Availability = [MTTF/(MTTF + MTTR)] ×100%

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

ii). Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

For example, some of the hazards associated with a computer-based cruise control for an automobile might be

- causes uncontrolled acceleration that cannot be stopped
- does not respond to depression of brake pedal (by turning off)
- does not engage when switch is activated
- slowly loses or gains speed

Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.To be effective, software must be analyzed in the context of the entire system.

If a set of external environmental conditions are met (and only if they are met), the improper position of the mechanical device will cause a disastrous failure. Analysis techniques such as *fault tree analysis* [VES81], *real-time logic* [JAN86], or *petri net models* [LEV87] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap.

THE ISO 9000 QUALITY STANDARDS:-

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management

ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001:2000 standard is applicable to all engineering disciplines, a special set of ISO guidelines have been developed to help interpret the standard for use in the software process.

The requirements delineated by ISO 9001 address topics such as

- management responsibility,
- quality system, contract review,
- design control,
- document and data control,
- product identification and traceability,
- process control,
- inspection and testing,
- corrective and preventive action,
- control of quality records,
- internal quality audits,
- training,
- servicing and
- statistical techniques.

In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed.