NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India

NRCM
Your roots to success...

UGC - Autonomous Institute
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH

INFORMATION TECHNOLOGY

B-tech III Year I Semester								
Course Code	**Category**	**Hours/Weak**			**Credits**	**Max Marks**		
		L	**T**	**P**	**C**	**CIE**	**SEE**	**Total**
23IT501	**Professional Core**	**3**	**0**	**0**	**3**	**40**	**60**	**100**
Contact Classes:51	**Tutorial classes: Nil**	**Practical classes: Nil**				**Total Classes:51**		

1. Ability to translate end-user requirements into system and software requirements, using e.g. UML, and structure the requirements in a Software Requirements Document (SRD).
2. Identify and apply appropriate software architectures and patterns to carry out high level design of a system and be able to critically compare alternative choices.
3. Will have experience and/or awareness of testing problems and will be able to develop a simple testing report

UNIT - I

Introduction to Software Engineering: The evolving role of software, changing nature of software,
software myths. **A Generic view of process**: Software engineering- a layered technology, a process
framework, the capability maturity model integration (CMMI). **Process models**: The waterfall model,
Spiral model and Agile methodology

UNIT – II

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.

UNIT – III

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, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams.

UNIT - IV

Testing Strategies: A strategic approach to software testing, test strategies for conventional software, black-box and white-box testing, validation testing, system testing, the art of debugging. Metrics for Process and Products: Software measurement, metrics for software quality.

UNIT – V

Risk management: Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM. **Quality Management:** Quality concepts, software quality assurance, software reviews, formal technical reviews, statistical software quality assurance, software reliability, the ISO 9000 quality standards.

TEXT BOOKS:

1. Software Engineering, A practitioner's Approach- Roger S. Pressman, 6th edition, McGraw Hill International Edition.
2. Software Engineering- Sommerville, 7th edition, Pearson Education.

REFERENCE BOOKS:

1. The unified modeling language user guide Grady Booch, James Rambaugh, Ivar Jacobson, Pearson Education.
2. Software Engineering, an Engineering approach- James F. Peters, Witold Pedrycz, John Wiley.
3. Software Engineering principles and practice- Waman S Jawadekar, The McGraw-Hill Companies.
4. Fundamentals of object-oriented design using UML Meiler page-Jones: Pearson Education.

UNIT-I

1.1 What is Software Engineering?

1. The term software engineering is the product of two words, software, and engineering.
2. The software is a collection of integrated programs.

Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages. Computer programs and related documentation such as requirements, design models and user manuals.

Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc.

Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Why is Software Engineering required?

Software Engineering is required due to the following reasons:

1. To manage Large software
2. For more Scalability
3. Cost Management
4. To manage the dynamic nature of software
5. For better quality Management

Need of Software Engineering

The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.

1. **Huge Programming:** It is simpler to manufacture a wall than to a house or building, similarly, as the measure of programming become extensive engineering has to step to give it a scientific process.
2. **Adaptability:** If the software procedure were not based on scientific and engineering ideas, it would be simpler to re-create new software than to scale an existing one.
3. **Cost:** As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the proper process is not adapted.
4. **Dynamic Nature:** The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one.
5. **Quality Management:** Better procedure of software development provides a better and
6. quality software product.

1.What is software engineering?

1. Your thoughts here

1.Related to the process: a systematic procedure used for the analysis, design, implementation, test and maintenance of software.

2.Related to the product: the software should be efficient, reliable, usable, modifiable, portable, testable, reusable, maintainable, interoperable, and correct.

2. *The definition in IEEE Standard*:

1.Theapplicationofasystematic,disciplined,quantifiableapproachtothed evelopment, operation, and maintenance of software, that is, the application of engineering to software.

2.Thestudyofapproachesasin1993:TheJointIEEEComputerSocietyand ACMSteering Committee for the establishment of software engineering as a profession.

1.2 The Evolving Role of Software:

1. *Software is a product*
   1. Delivers computing potential
   2. Produces,manages,acquires,modifies,displays,or transmits information.

2. *Software is a vehicle for delivering a product*
   1. Supports or directly provides system functionality
   2. Controls other programs(e.g.,an operating system)
   3. Effects communications(e.g.,networking software)
   4. Helps build other software(e.g.,softwaretools)

3. The Law of Continuing Change (1974):E-type systems must be continually adapted else they become progressively less satisfactory.

4. The Law of Increasing Complexity(1974):As an E-Type system evolves its complexity increases unless work is done to maintain or reduce it.

5. The Law of Self Regulation (1974):The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

6. The Law of Conservation of Organizational Stability(1980):The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

7. The Law of Conservation of Familiarity (1980): As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution.

8. The Law of Continuing Growth (1980):The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

9. The Law of Declining Quality (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

10. The Feedback SystemLaw (1996):E-typeevolutionprocessesconstitutemulti-level,multi-loop,multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

1.3 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

1. heavy interaction with computer hardware
2. heavy usage by multiple users
3. concurrent operation that requires scheduling, resource sharing, and sophisticated process management
4. complex data structures
5. multiple external interfaces

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

6. Application software:

1. Application software consists of standalone programs that solve a specific business need.
2. It facilitates business operations or management/technical decision making.
3. It is used to control business functions in real-time
4. *E.g.* point-of-sale transaction processing, real-time manufacturing process control.

5. **Engineering/Scientific software:**

1. Engineering and scientific applications range
2. from astronomy to volcanology
3. from automotive stress analysis to space shuttle orbital dynamics
4. from molecular biology to automated manufacturing

E.g. computer aided design, system simulation and other interactive applications.

5. Embedded software:

1. 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.
2. 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.

3. **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

4. **Web-applications:** Web Apps 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.

5. **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:

1. **Ubiquitous computing**
2. **Netsourcing**
3. **Open source**
4. **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.

1.4 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.

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.

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.

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.

2.A GENERIC VIEW OF PROCESS

1. SOFTWAREENGINEERING-A LAYERED TECHNOLOGY

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. **The bedrock that supports software engineering is a quality focus.**

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers. **Process defines a framework that must be established for effective delivery of software engineering technology**.

The software forms the basis for management control of software projects and establishes the context in which

1. Technical methods are applied,
2. Work products are produced,
3. Milestones are established,
4. Quality is ensured,
5. And change is properly managed.

Software engineering methods rely on a set of basic principles that govern area of the technology and include modeling activities.

Methods encompass a broad array of tasks that include

1. Communication,
2. Requirements analysis,
3. Design modeling,
4. Program construction,
5. Testing and support.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

2.2A PROCESS FRAMEWORK

1. **Software process** must be established for effective delivery of software engineering technology.
2. A **process framework** establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
3. The process framework encompasses a **set of umbrella activities** that are applicable acrossthe entire software process.
4. Each **framework activity** is populated by a set of software engineering actions
5. Each **software engineering action** is represented by a number of different task sets- each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

In brief

"A **process** defines who is doing what, when, and how to reach a certain goal." A **Process Framework**

1. Establishes the foundation for a complete software process
2. Identifies a small number of **framework activities**
3. Applies to all s/w projects, regardless of size/complexity.
4. Also, set of **umbrella activities**
5. Applicable across entire s/w process.

Each **framework activity** has

1.Set of **s/w engineering actions**.
2.Each **s/w engineering action** (e.g., design) has
3.Collection of related **tasks (**called **task sets):**
4.Work tasks
5.Work products (deliverables)
6.Quality assurance points
7. Project milestones.

Generic Process Framework: It is applicable to the vast majority of software projects

1. Communication activity
2. Planning activity
3. Modelling activity
4. Analysis action
5. Requirements gathering work task
6. Elaboration work task
7. Negotiation work task
8. Specification work task
9. Validation work task
10. Design action
11. Data design work task
12. Architectural design work task
13. Interface design work task
14. Component-level design work task
15. Construction activity
16. Deployment activity

1. Communication: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.

2.Planning: This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

3.Modeling: This activity encompasses the creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements. The modelling activity is composed of 2 software engineering actions- analysis and design.

i) Analysis encompasses a set of work tasks.

ii)Design encompasses work tasks that create a design model.

4.Construction: This activity combines core generation and the testing that is required to uncover the errors in the code.

5.Deployment: The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evolution.

These 5 generic framework activities can be used during the development of small programs, the creation of large web applications, and for the engineering of large, complex computer-based systems.

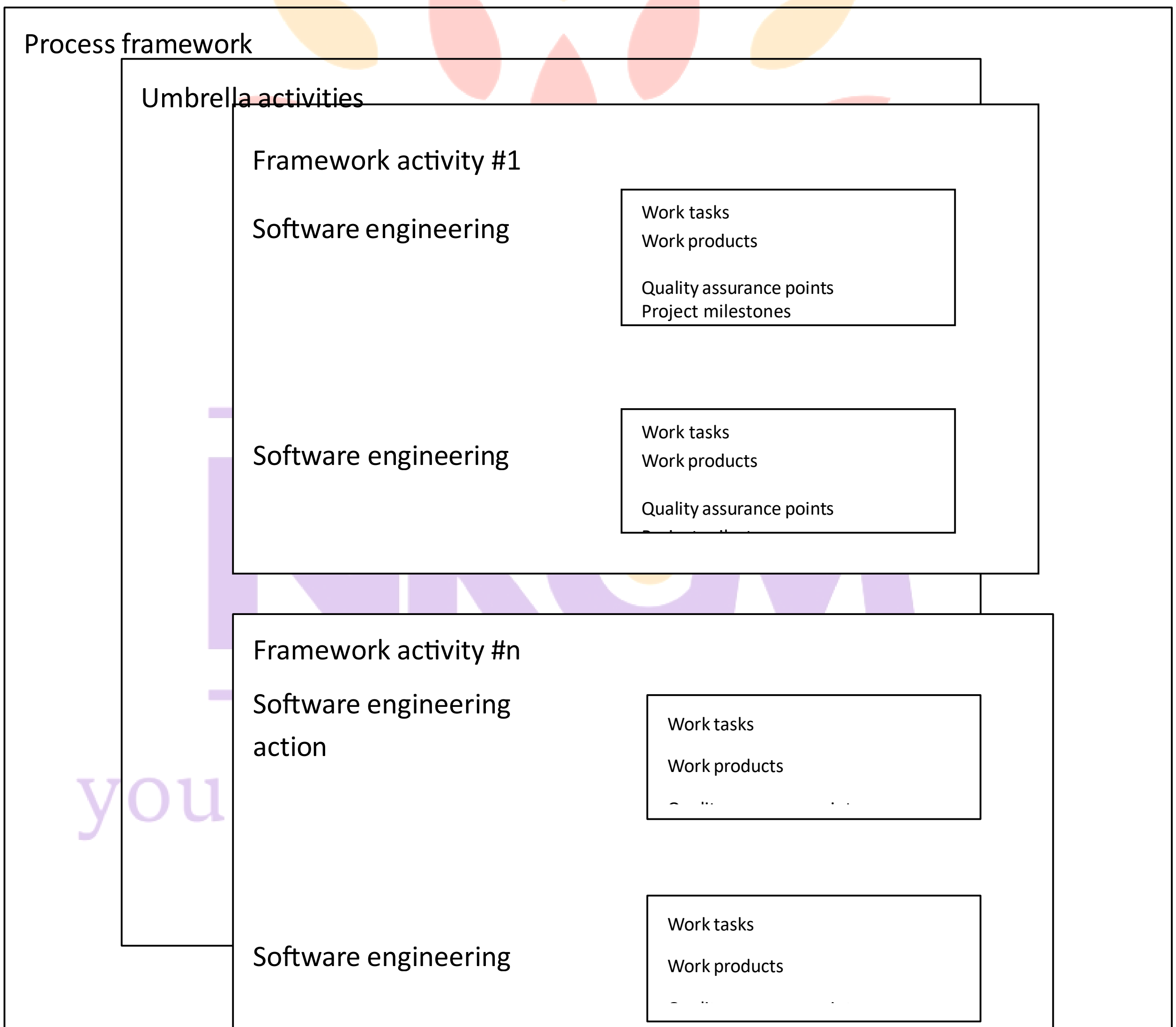
The following are the set of **Umbrella Activities.**

1. **Software project tracking and control** – allows the software team to assess progress against the project plan and take necessary action to maintain schedule.
2. **Risk Management -** assesses risks that may effect the outcome of the project or the quality of the product.
3. **Software Quality Assurance -** defines and conducts the activities required to ensure software quality.
4. **Formal Technical Reviews** - assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.
5. **Measurement -** define and collects process, project and product measures that assist the team in delivering software that needs customer's needs, can be used in conjunction with all other framework and umbrella activities.
6. **Software configuration management -** manages the effects of change throughout the software process.
7. **Reusability management** - defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
8. **Work Product preparation and production -** encompasses the activities required to create work products such as models, document, logs, forms and lists.

Intelligent application of any software process model must recognize that adaption is essential for success but process models do differ fundamentally in:

1. The overall flow of activities and tasks and the interdependencies among activities and tasks.
2. The degree through which work tasks are defined within each frame work activity.
3. The degree through which work products are identified and required.
4. The manner which quality assurance activities are applied.
5. The manner in which project tracking and control activities are applied.
6. The overall degree of the detailed and rigor with which the process is described.
7. The degree through which the customer and other stakeholders are involved with the project.
8. The level of autonomy given to the software project team.
9. The degree to which team organization and roles are prescribed.

2.3THE CAPABILITY MATURITY MODEL INTEGRATION(CMMI)



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

1. As a continuous model
2. As a staged model.

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

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

3.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.

Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality software.

Communication

1. Involves communication among the customer and other stake holders.
2. It encompasses requirements gathering.

Planning

1. Establishes a plan for software engineering work
2. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry.
3. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.
4. Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage.
5. It addresses technical tasks, resources, work products and work schedule

Modelling (Analyze and Design)

6. Requirement analysis is the most important and fundamental stage
7. A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any).
8. The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS - Design Document Specification.

Construction (Code and Test)

9. In this stage of SDLC the actual development starts and the product is built.

10. Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code.
11. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

12. The testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS - Software Requirement Specification.

Deployment

13. Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).
14. Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

3.1THE 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.

The picture can't be displayed.

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.

3.2THE SPIRAL MODEL

1. 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.
2. The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.
3. 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.

The picture can't be displayed.

1. **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.

2. 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.

3. 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.

4. 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.

1. 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.

2. 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.

3. 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.

3.3 Agile Methodology

The meaning of Agile is swift or versatile."**Agile process model**" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

Phases of Agile Model:

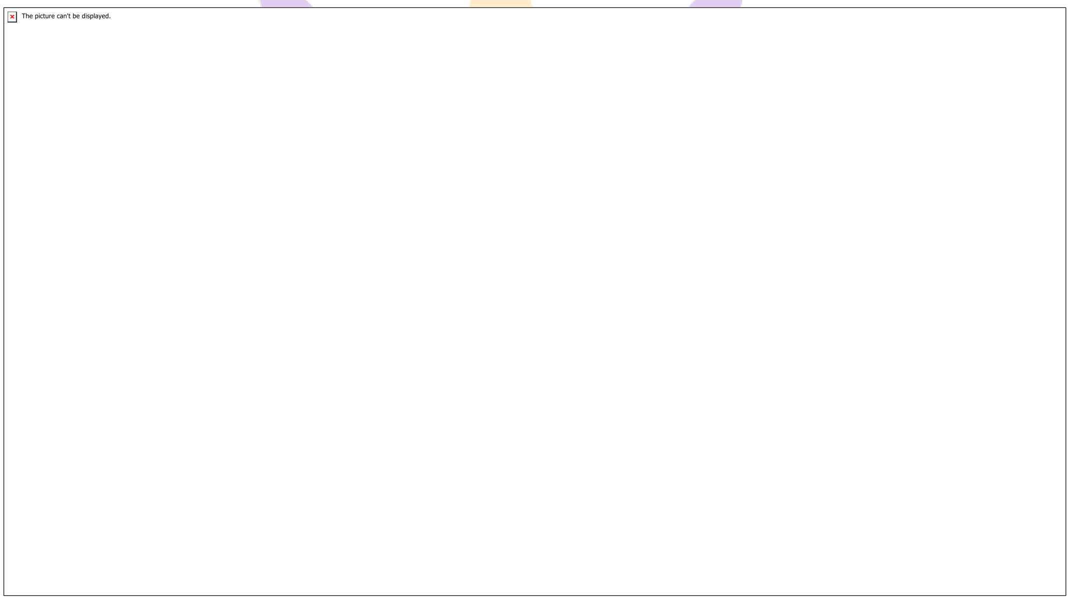
Following are the phases in the Agile model are as follows:

1. Requirements gathering
2. Design the requirements
3. Construction/ iteration

4. Testing/ Quality assurance
5. Deployment
6. Feedback

1.Requirements gathering: In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.

2. Design the requirements: When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.



3. Construction/ iteration: When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.

4. Testing: In this phase, the Quality Assurance team examines the product's performance and looks for the bug.

5. Deployment: In this phase, the team issues a product for the user's work environment.

6. Feedback: After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Agile Testing Methods:

1. Scrum

2. Crystal
3. Dynamic Software Development Method(DSDM)
4. Feature Driven Development(FDD)
5. Lean Software Development
6. eXtreme Programming(XP)

Scrum

SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.

There are three roles in it, and their responsibilities are:

1. Scrum Master: The scrum can set up the master team, arrange the meeting and remove obstacles for the process
2. Product owner: The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.
3. Scrum Team: The team manages its work and organizes the work to complete the sprint or cycle.

extreme Programming(XP)

This type of methodology is used when customers are constantly changing demands or requirements, or when they are not sure about the system's performance.

Crystal:

There are three concepts of this method-

1. Chartering: Multi activities are involved in this phase such as making a development team, performing feasibility analysis, developing plans, etc.
2. Cyclic delivery: under this, two more cycles consist, these are:
    1. Team updates the release plan.
    2. Integrated product delivers to the users.
3. Wrap up: According to the user environment, this phase performs deployment, post-deployment.

Dynamic Software Development Method(DSDM):

DSDM is a rapid application development strategy for software development and gives an agile project distribution structure. The essential features of DSDM are that users must be actively connected, and teams have been given the right to make decisions. The techniques used in DSDM are:

1. Time Boxing
2. MoSCoW Rules

3. Prototyping

The DSDM project contains seven stages:

1. Pre-project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design and build Iteration
6. Implementation
7. Post-project

Feature Driven Development (FDD):

This method focuses on "Designing and Building" features. In contrast to other smart methods, FDD describes the small steps of the work that should be obtained separately per function.

Lean Software Development:

Lean software development methodology follows the principle "just in time production." The lean method indicates the increasing speed of software development and reducing costs. Lean development can be summarized in seven phase.

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.

System models: Context models, behavioral models, data models, object models, structured methods.

SOFTWARE REQUIREMENTS

Software requirements are necessary

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

What is a requirement?

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

Requirements engineering:

1. The process of finding out, analysing documenting and checking these services and constraints is called requirement engineering.
2. The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
    1. 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 organisation’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:

1. **User requirements**

Statements in natural language plus diagrams ofthe services the system provides and its operational constraints. Written for customers.

2. **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.

User Requirement Definition:

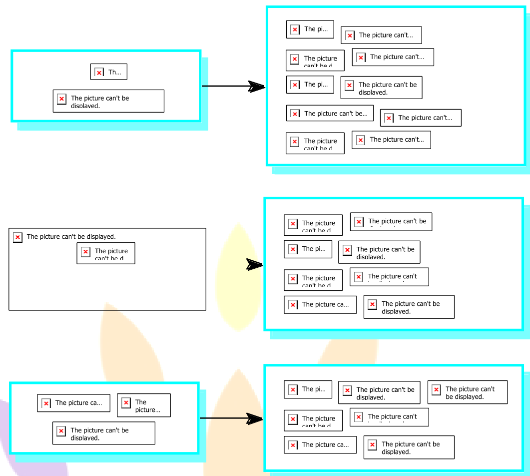
The software must provide the means of representing and accessing external files created by other

tools.

System Requirement specification:

1. The user should be provided with facilities to define the type of external files.
2. Each external file type may have an associated tool which may be applied to the file.
3. Each external file type may be represented as a specific icon on the user's display.
4. Facilities should be provided for the icon representing an external file type to be defined bythe user.
5. When a 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.

Requirements readers:



1.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 particularsituations.

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.

1.1 FUNCTIONAL REQUIREMENTS:

Describe functionality or system services.

Depend on the type of software, expected users and the type of system where the softwareis 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

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

Requirements imprecision

1. Problems arise when requirements are not precisely stated.
2. Ambiguous requirements may be interpreted in different ways by developers and users.
3. Consider the term 'appropriate viewers'
4. User intention - special purpose viewer for each different document type;
5. Developer interpretation - Provide a text viewer that shows the contents of the document.

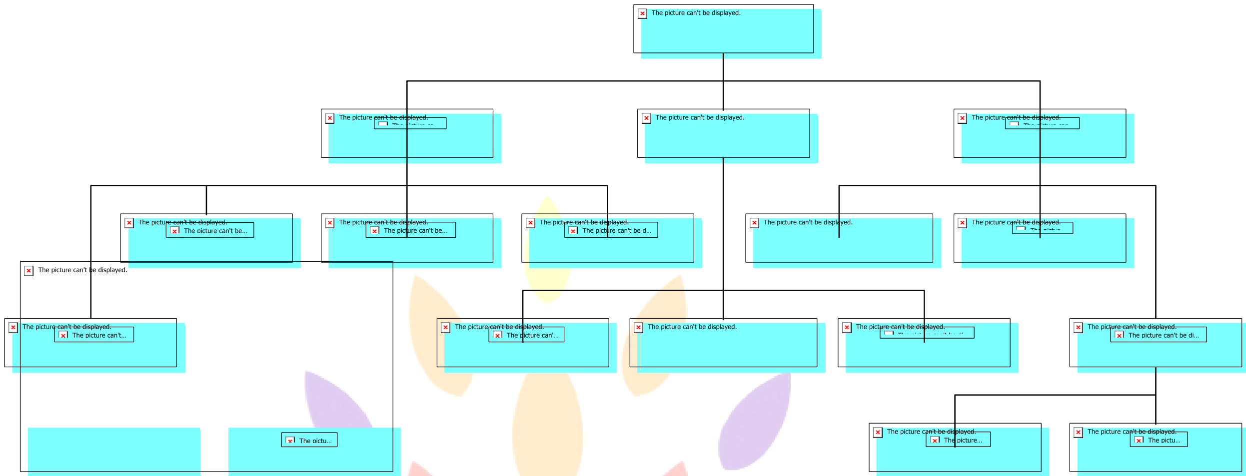
Requirements completeness and consistency:

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

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



Non-functional requirement types:

Non-functional requirements: Product requirements

1. Requirements which specify that the delivered product must behave in a particular way
   e.g. execution speed, reliability, etc.

USER REQUIREMENTS

2. Should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technicalknowledge.
3. 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

1. Precision is difficult without making the document difficult to read.
   Requirements confusion
2. Functional and non-functional requirements tend to bemixed-up.
   Requirements amalgamation
3. 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

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

Guidelines for writing requirements

Invent a standard format and use it for all requirements.

1. Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
2. Use text highlighting to identify key parts of the requirement.
3. Avoid the use of computer jargon.

SYSTEM REQUIREMENTS

1. More detailed specifications of system functions, services and constraints than userrequirements.
2. They are intended to be a basis for designing the system.
3. They may be incorporated into the system contract.
4. 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

1. A system architecture may be designed to structure the requirements;
2. The system may inter-operate with other systems that generate design requirements;
3. 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 modularization.
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 thesystem. 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

1. The freedom of the requirements writer is limited by a predefined template for requirements.
2. All requirements are written in a standard way.
3. The terminology used in the description may belimited.

4. 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

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

Tabular specification

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

Graphical models

13. 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

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

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 beingspecified
2. Description of its inputs and where these comefrom
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.

1. INTERFACE SPECIFICATION

1. Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
2. Three types of interface may have to be defined

1. ***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)
2. ***Data structures that are exchanged*** **t**hat are passed from one sub-system to another. Graphical data models are the best notations for this type of description
3. ***Data representations*** that have been established for an existing sub-system

3. Formal notations are an effective technique for interface specification.

4. THE SOFTWARE REQUIREMENTS DOCUMENT:

1. The requirements document is the official statement of what is required of the system developers.

2. Should include both a definition of user requirements and a specification of the system requirements.

3. It is NOT a design document. As far as possible, it should set of WHAT the system shoulddo rather than HOW it should do it

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.
1. Purpose of the requirements document
2. Scope of the project
3. Definitions, acronyms and abbreviations
4. References
5. Overview of the remainder of the document
2. General description.
1. Product perspective
2. Product functions
3. User characteristics
4. General constraints
5. 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.

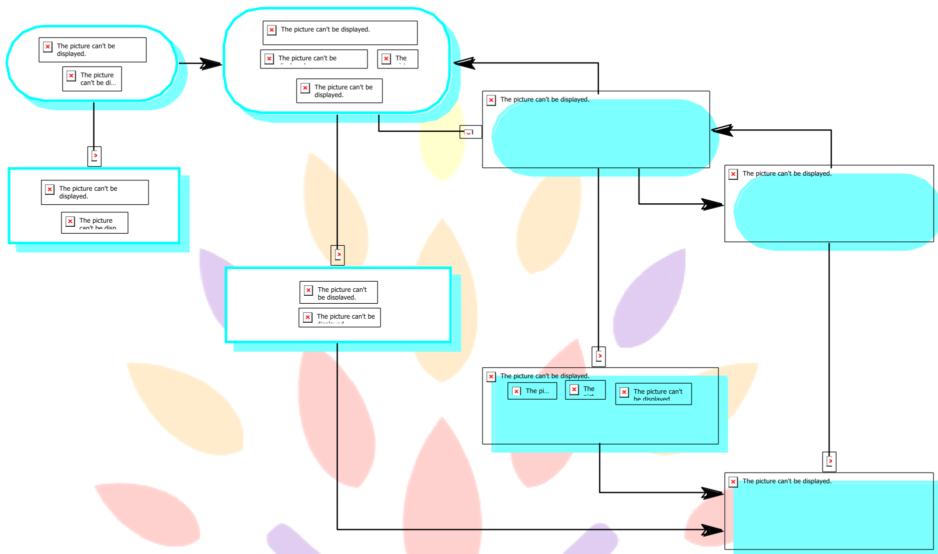
5.Index.

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

1. Assessing whether the system is useful to the business(feasibility study)
2. Discovering requirements(elicitation and analysis)
3. Converting these requirements into some standard form(specification)
4. Checking that the requirements actually define the system that the customerwants(validation) The process of managing the changes in the requirements is called **requirementmanagement**.

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 systemmodeling.

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.

1. 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.

1.A short focused study that checks
1.If the system contributes to organizational objectives;
2.If the system can be engineered using current technology and withinbudget;
3.If the system can be integrated with other systems that are used.

Feasibility study implementation:

2. A feasibility study involves information assessment, information collection and report writing.
3. Questions for people in the organization
1.What if the system wasn't implemented?
2.What are current process problems?
3.How will the proposed system help?
4.What will be the integration problems?
5.Is new technology needed? What skills?
6. 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.

4.REQUIREMENT ELICITATION AND ANALYSIS:

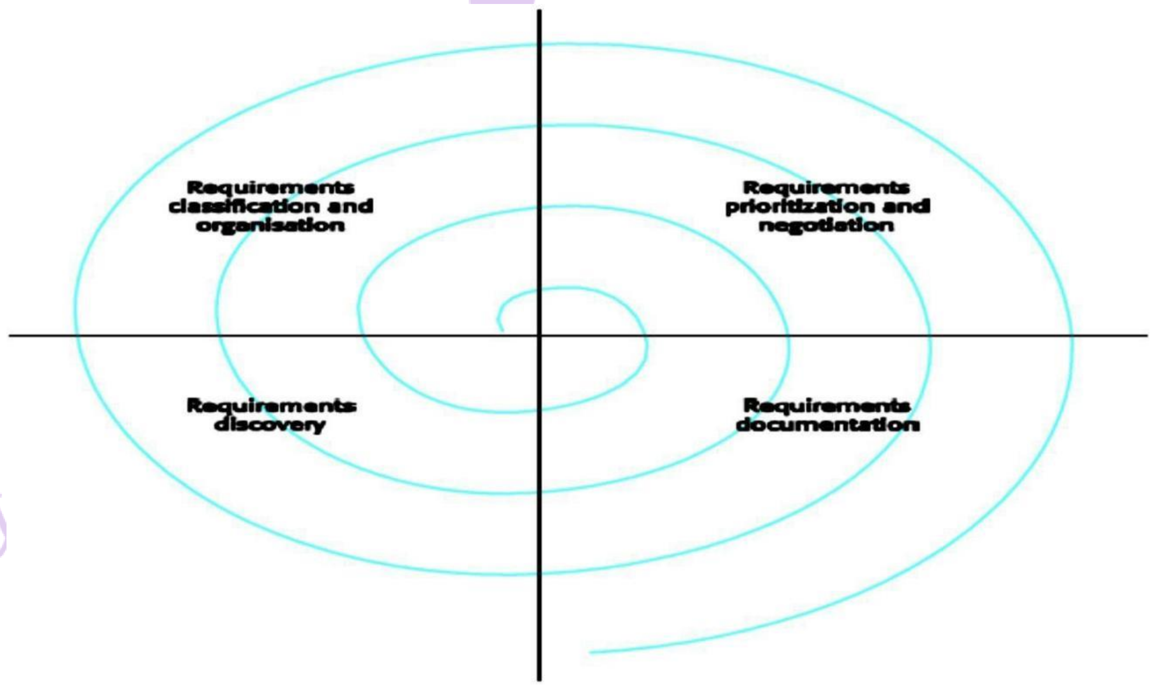
The requirement engineering process is requirements elicitation and analysis.

5. Sometimes called requirements elicitation or requirements discovery.
6. 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.
7. May involve end-users, managers, engineers involved in maintenance, domainexperts, trade unions, etc. These are called *stakeholders.*

Problems of requirements analysis

8. Stakeholders don't know what they really want.
9. Stakeholders express requirements in their own terms.
10. Different stakeholders may have conflicting requirements.
11. Organizational and political factors may influence the system requirements.
12. The requirements change during the analysis process. New stakeholders mayemerge and the business environment change.

The requirements spiral



Process activities

1. Requirements discovery

Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

2. Requirements classification and organization

Groups related requirements and organizes them into coherent clusters.

3. Prioritization and negotiation

Prioritizing 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.

5.REQUIREMENTS VALIDATION

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

Requirements checking:

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

Requirements validation techniques

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

Requirements reviews:

1. Regular reviews should be held while the requirements definition is being formulated.
2. Both client and contractor staff should be involved in reviews.
3. Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an earlystage.

Review checks:

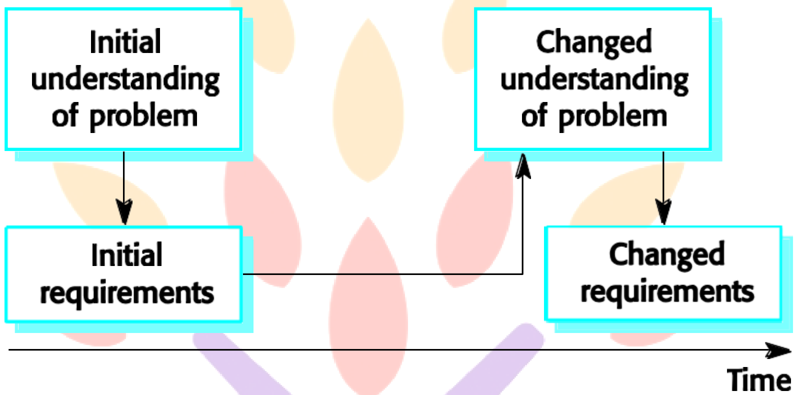
1. ***Verifiability***: Is the requirement realistically testable?
2. ***Comprehensibility***: Is the requirement properly understood?
3. ***Traceability***: Is the origin of the requirement clearly stated?
4. ***Adaptability***: Can the requirement be changed without a large impact on other requirements?

5.REQUIREMENTS MANAGEMENT

1. Requirements management is the process of managing changing requirements duringthe requirements engineering process and system development.
2. Requirements are inevitably incomplete and inconsistent
3. New requirements emerge during the process as business needs change and abetter understanding of the system is developed;
4. Different viewpoints have different requirements and these are oftencontradictory.
5. **Requirements change**
6. The priority of requirements from different viewpoints changes during the developmentprocess.

7. System customers may specify requirements from a business perspective that conflict withend-user requirements.
8. The business and technical environment of the system changes during its development.

Requirements evolution:



Enduring and volatile requirements:

1. ***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
2. ***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.

Requirements management planning:

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

Traceability:

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

1. Source traceability

2. Links from requirements to stakeholders who proposed these requirements;
3. Requirements traceability
4. Links between dependent requirements;
5. Design traceability - Links from the requirements to the design;

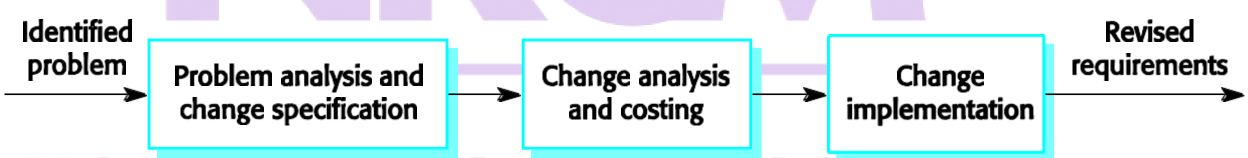
CASE tool support:

1. Requirements storage
2. Requirements should be managed in a secure, managed data store.
3. Change management
4. The process of change management is a workflow process whose stages can bedefined and information flow between these stages partially automated.
5. Traceability management
6. –Automated retrieval of the links between requirements.

Requirements change management:

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

Change management:



UNIT-III

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, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams.

Design Engineering

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

1. Design principles establish an overriding philosophy that guides the designer in the work that is performed.
2. Design concepts must be understood before the mechanics of design practice are applied and
3. 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.

1. DESIGN PROCESS AND DESIGN QUALITY:

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

Goals of design:

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

1. 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.
2. The design must be a readable, understandable guide for those who generate code and forthose who test and subsequently support the software.
3. 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 has been created using recognizable architectural styles or patterns is composed of components that exhibit good design characteristics and 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 andcomponents.
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.

2. 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.

1. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
1. 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; chamges 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.

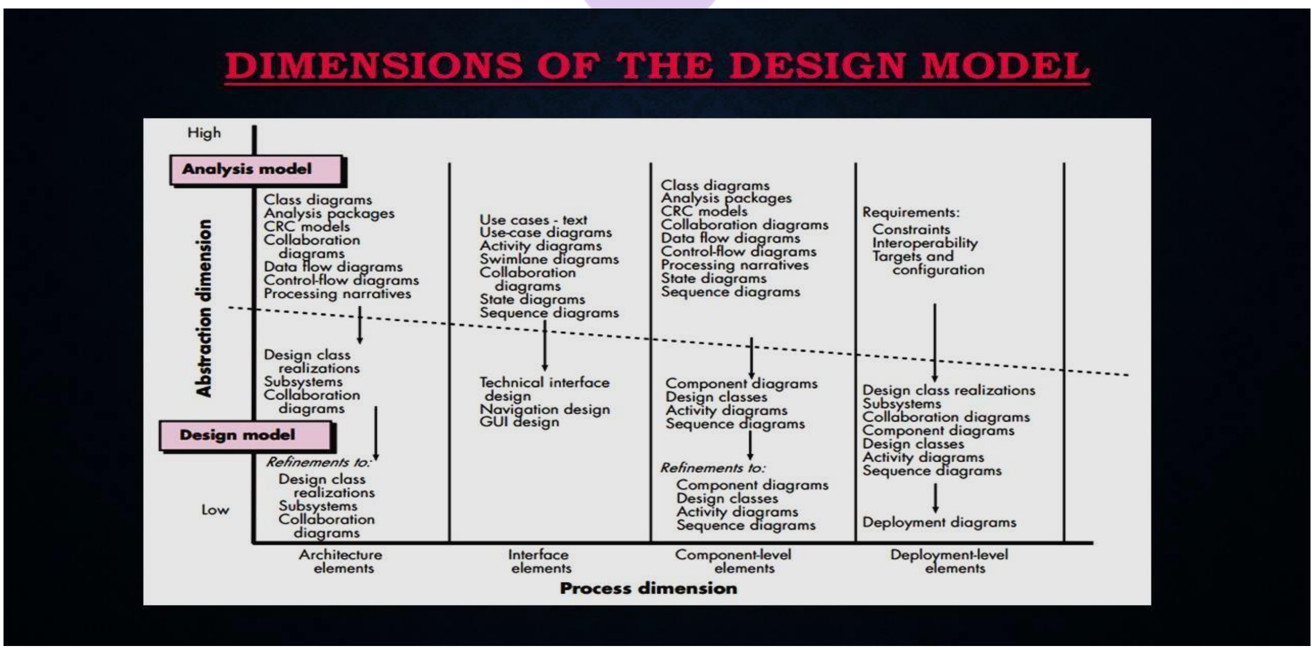
THE DESIGN MODEL:

4. The design model can be viewed into different dimensions.
5. 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.



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.

1. 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.
2. At **the application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
3. 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.

1. 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

4. Interface design elements:

The *interface design* for software is the equivalent to a set of detailed drawings for thedoors, 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:

The user interface(UI);

1. External interfaces to other systems, devices, networks, or other produces or consumers of information; and Internal interfaces between various design components.
2. These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

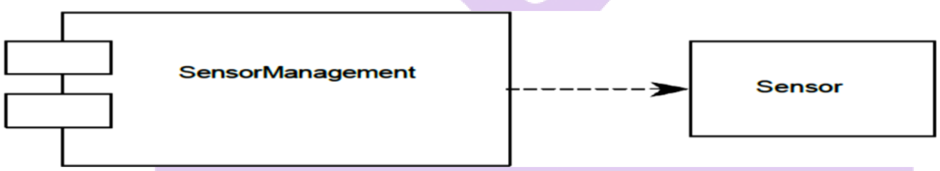
UI design is a major software engineering action.

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.

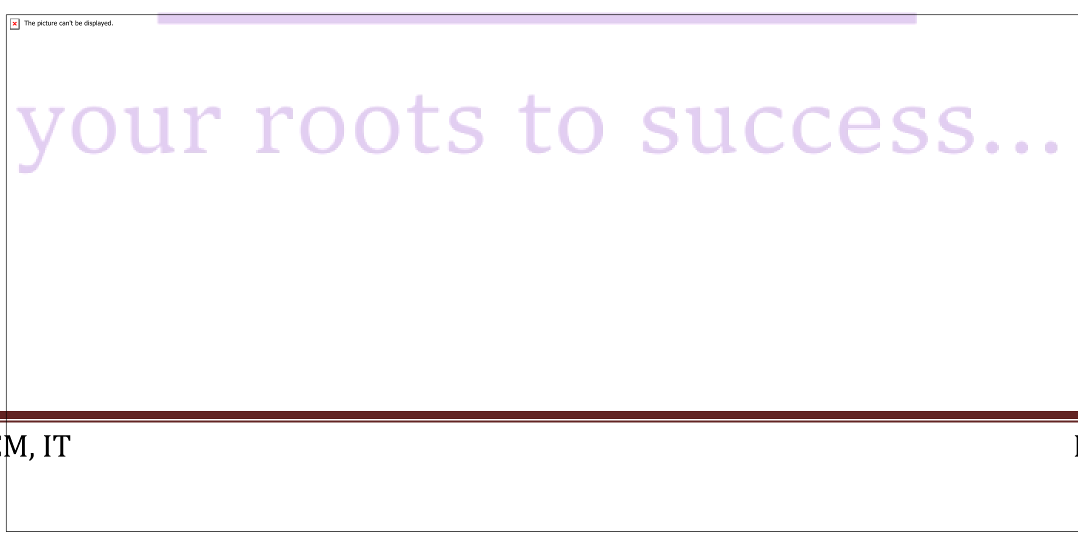
The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

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."

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.



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.



ARCHITECTURAL DESIGN

1.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

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

1. The architecture is a representation that enables a software engineer to 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,
reducing the risks associated with the construction of the software.

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

1. data design
1. architectural design.
2. Data design enables us to represent the data component of the architecture.
3. 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:

1. Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
1. 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.
1. Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

2.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.

1. 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-qualityapplications.
1. 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.
1. 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.

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.

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 todata.
2. All data structures and the operations to be performed on each should beidentified.
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 designprocess.
5. The representation of data structure should beknown 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:

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

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

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

Constraints that define how components can be integrated to form the system; and

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

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:

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

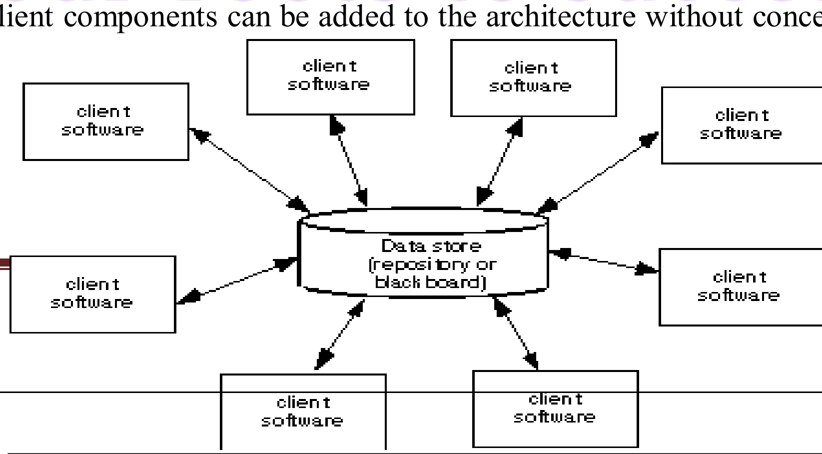
A pattern imposes a rule on the architecture, describing how the software will handle someaspect of its functionality at the infrastructure level.

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

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

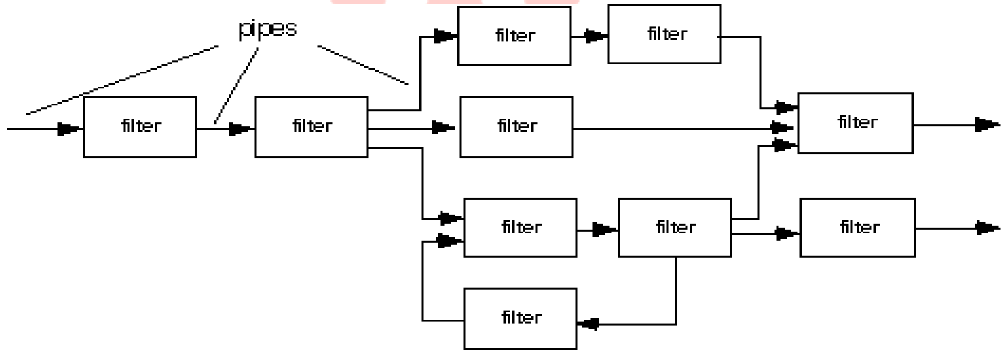
Data-centered architectures promote ***integrability***. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients



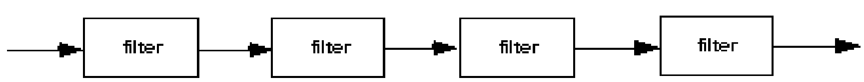
(because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism

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

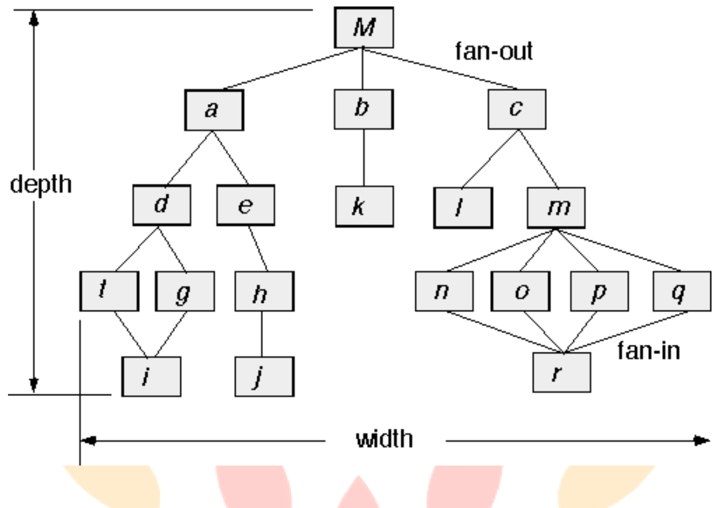


(b) batch sequential

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:

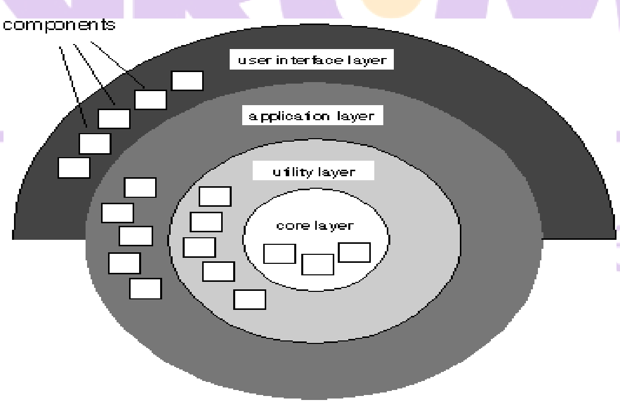
1. ***Main program/subprogram architectures***. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.

2. ***Remote procedure call architectures***. The components of a main program/ subprogram architecture are distributed across multiple computers on a network



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.



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 someaspect 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

1. *operating system process management* pattern
2. *task scheduler* pattern

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

3.a ***database management system*** pattern that applies the storage and retrieval capability of a DBMS to the application architecture
4.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

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

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.

1. How is control managed within the architecture?
2. Does a distinct control hierarchy exist, and if so, what is the role ofcomponents within this control hierarchy?
3. How do components transfer control within the system?
4. How is control shared among components?

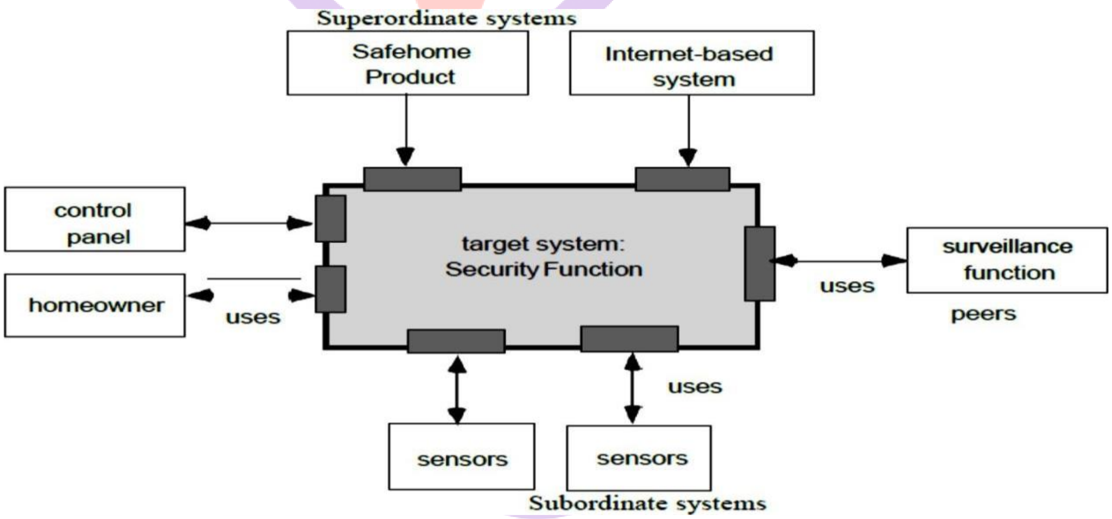
Data.

5.How are data communicated between components?
6.Is the flow of data continuous, or are data objects passed to the system sporadically?
7.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)?
8.Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?
9.How do functional components interact with data components?
10. 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:

1.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

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.

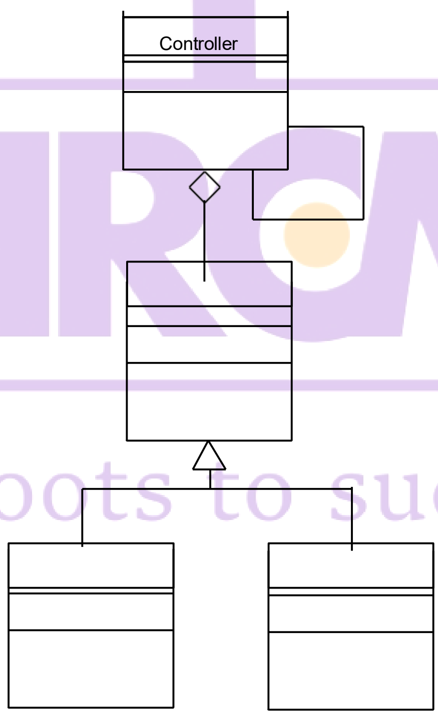
Detector: An abstraction that encompasses all sensing equipment that feeds information into the target system

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.

communicates with

Node



Detector Indicator

Figure 10.7 UML relat ionships for SafeHome security function archetypes (adapted f rom [BOS00])

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:

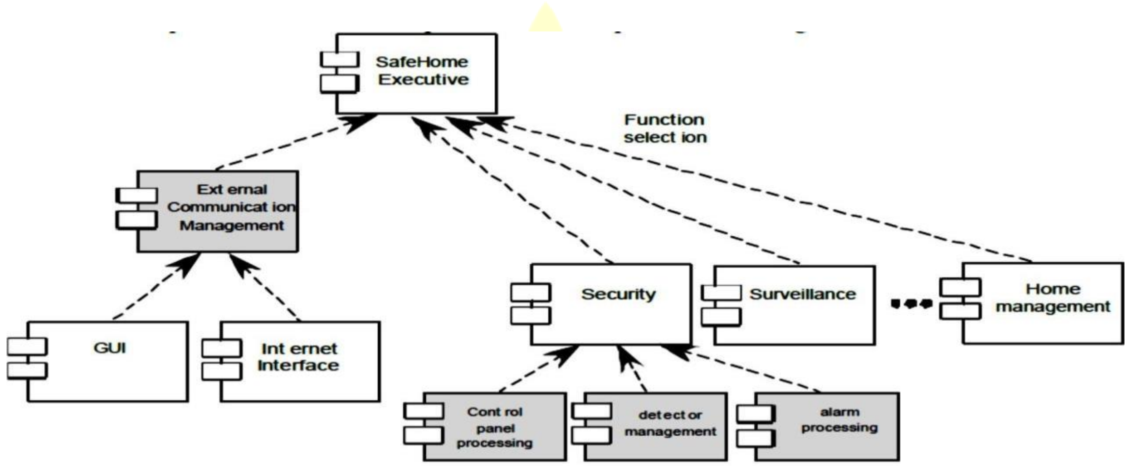
External communication management- coordinates communication of the security function with external entities

Control panel processing- manages all control panel functionality.

Detector management- coordinates access to all detectors attached to the system.

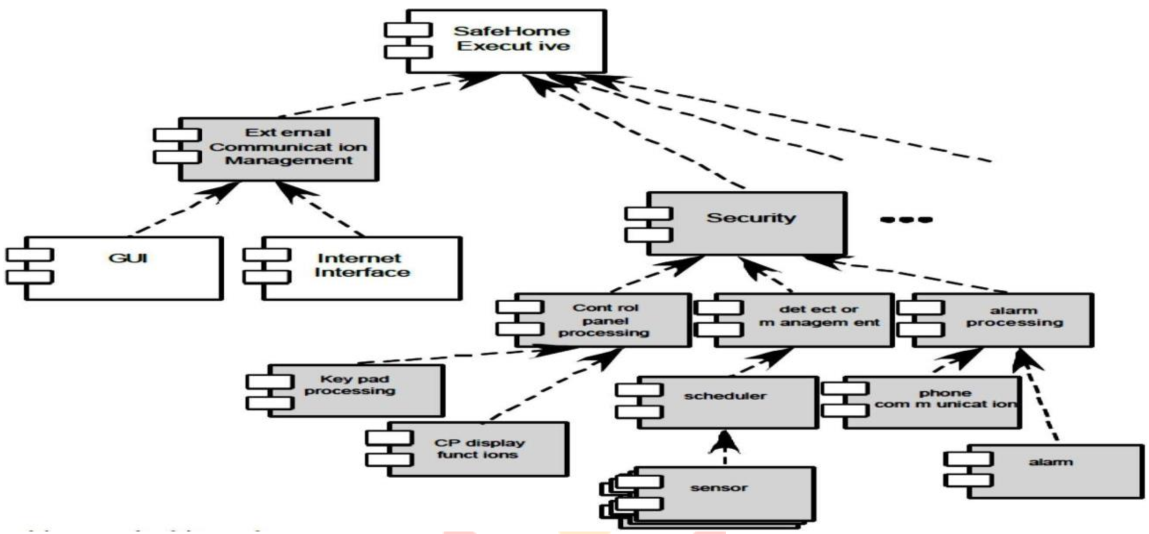
Alarm processing- verifies and acts on all alarm conditions.

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.



Component Structure

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 structureand components are appropriate.



A Conceptual Model of UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group.

1. UML stands for **Unified Modeling Language**.
2. UML is different from the other common programming languages such as C++, Java, COBOL, etc.
3. UML is a pictorial language used to make software blueprints.
4. UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.

To understand the conceptual model of UML, first we need to clarify what is a conceptual model? and why a conceptual model is required?

5.A conceptual model can be defined as a model which is made of concepts and their relationships.
6.A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.
As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

7.UML building blocks
8.Rules to connect the building blocks
9.Common mechanisms of UML

Object-Oriented Concepts

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

Following are some fundamental concepts of the object-oriented world −

10. **Objects** − Objects represent an entity and the basic building block.
11. **Class** − Class is the blue print of an object.
12. **Abstraction** − Abstraction represents the behavior of an real world entity.
13. **Encapsulation** − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
14. **Inheritance** − Inheritance is the mechanism of making new classes from existing ones.
15. **Polymorphism** − It defines the mechanism to exists in different forms.

OO Analysis and Design

OO can be defined as an investigation and to be more specific, it is the investigation of objects. Design means collaboration of identified objects.

Thus, it is important to understand the OO analysis and design concepts. The most important purpose of OO analysis is to identify objects of a system to be designed. This analysis is also done for an existing system. Now an efficient analysis is only possible when we are able to start thinking in a way where objects can be identified. After identifying the objects, their relationships are identified and finally the design is produced.

The purpose of OO analysis and design can described as −

16. Identifying the objects of a system.
17. Identifying their relationships.
18. Making a design, which can be converted to executables using OO languages.

OO Analysis → OO Design → OO implementation using OO languages

There are three basic steps where the OO concepts are applied and implemented. The steps can be defined as

The above three points can be described in detail as −

19. During OO analysis, the most important purpose is to identify objects and describe them in a proper way. If these objects are identified efficiently, then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each and every object has some type of responsibilities to be performed. When these responsibilities are collaborated, the purpose of the system is fulfilled.
20. The second phase is OO design. During this phase, emphasis is placed on the requirements and their fulfilment. In this stage, the objects are collaborated according to their intended association. After the association is complete, the design is also complete.
21. The third phase is OO implementation. In this phase, the design is implemented using OO languages such as Java, C++, etc.

Role of UML in OO Design

UML is a modeling language used to model software and non-software systems. Although UML is used for non- software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but o bjects.

If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.

Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement. Before understanding the UML in detail, the OO concept should be learned properly. Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

Basic Structural Modeling

Structural modeling captures the static features of a system. They consist of the following −

22. Classes diagrams
23. Objects diagrams
24. Deployment diagrams
25. Package diagrams
26. Composite structure diagram
27. Component diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them.

The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Class Diagram:

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of objectoriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction. UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

28. Analysis and design of the static view of an application.
29. Describe responsibilities of a system.
30. Base for component and deployment diagrams.
31. Forward and reverse engineering.

How to Draw a Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

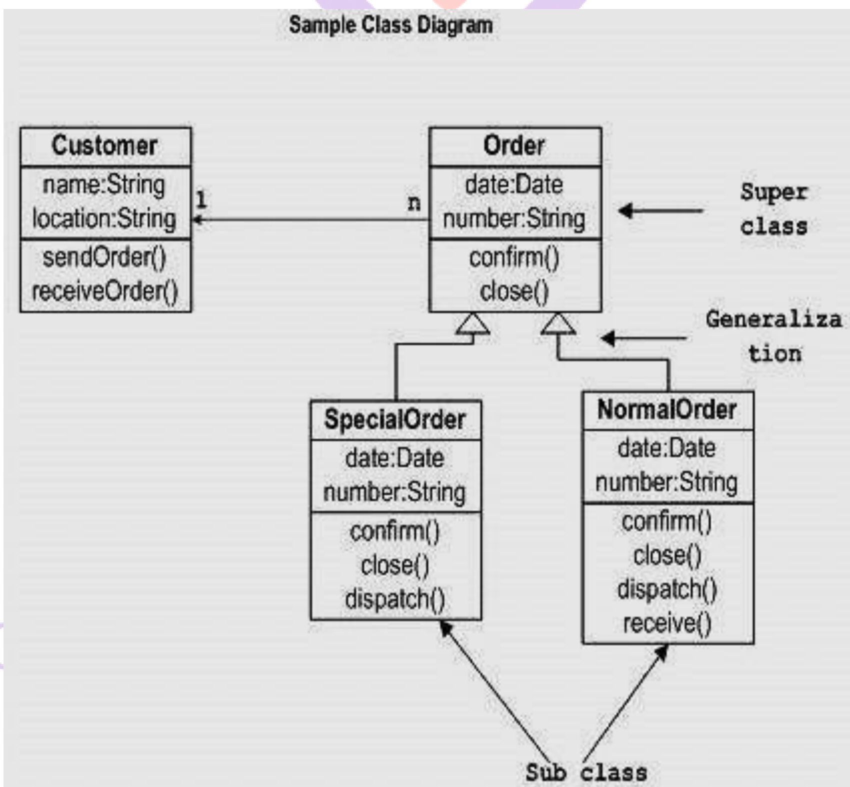
Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

The following points should be remembered while drawing a class diagram −

32. The name of the class diagram should be meaningful to describe the aspect of the system.
33. Each element and their relationships should be identified in advance.
34. Responsibility (attributes and methods) of each class should be clearly identified
35. For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
36. Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
37. Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.

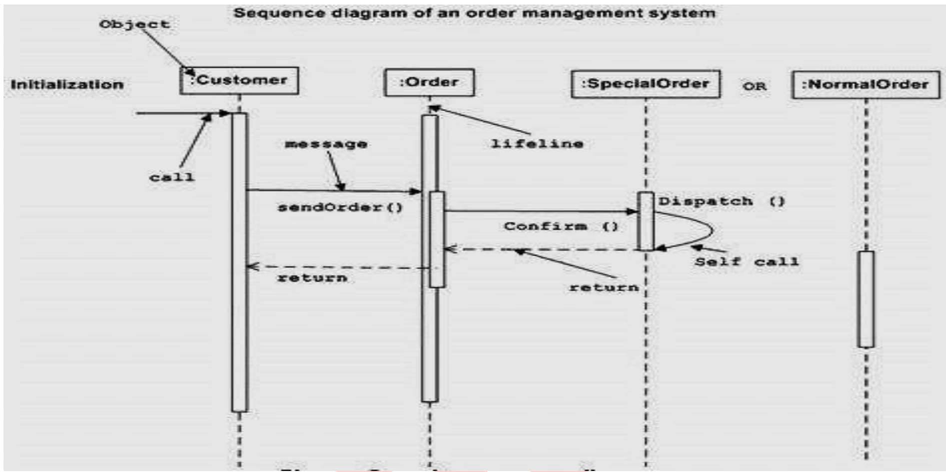
38. First of all, Order and Customer are identified as the two elements of the system. They have a one-to- many relationship because a customer can have multiple orders.
39. Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
40. The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().



The following class diagram has been drawn considering all the points mentioned above.

In a nutshell it can be said, class diagrams are used for −

41. Describing the static view of the system.
42. Showing the collaboration among the elements of the static view.
43. Describing the functionalities performed by the system.



44. Construction of software applications using object oriented languages.

The Sequence Diagram

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

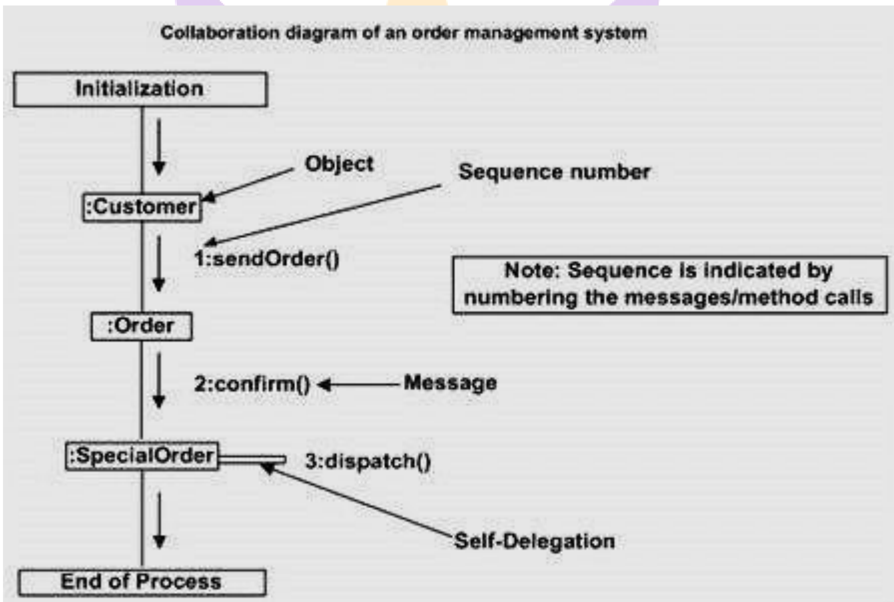
45. The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.

The Collaboration Diagram

46. The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

47. Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

48. To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



Use Case Diagram

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

Purpose of Use Case Diagrams

The main purpose of a use case diagram is to portray the dynamic aspect of a system. It accumulates the system's requirement, which includes both internal as well as external influences. It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams. It represents how an entity from the external environment can interact with a part of the system.

Following are the purposes of a use case diagram given below:

1.It gathers the system's needs.
2.It depicts the external view of the system.
3.It recognizes the internal as well as external factors that influence the system.
4.It represents the interaction between the actors.

How to draw a Use Case diagram?

It is essential to analyze the whole system before starting with drawing a use case diagram, and then the system's functionalities are found. And once every single functionality is identified, they are then transformed into the use cases to be used in the use case diagram.

After that, we will enlist the actors that will interact with the system. The actors are the person or a thing that invokes the functionality of a system. It may be a system or a private entity, such that it requires an entity to be pertinent to the functionalities of the system to which it is going to interact.

Once both the actors and use cases are enlisted, the relation between the actor and use case/ system is inspected. It identifies the no of times an actor communicates with the system. Basically, an actor can interact multiple times with a use case or system at a particular instance of time.

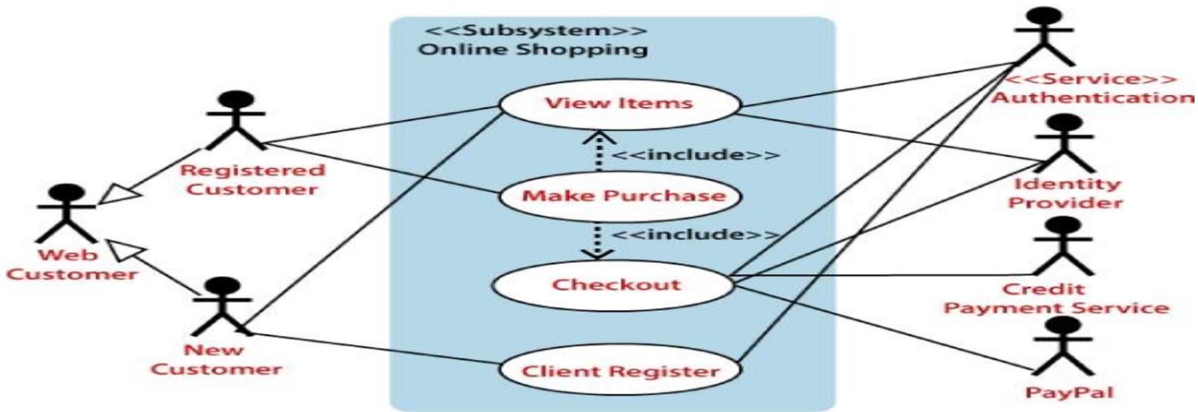
Following are some rules that must be followed while drawing a use case diagram:

1. A pertinent and meaningful name should be assigned to the actor or a use case of a system.
2. The communication of an actor with a use case must be defined in an understandable way.
3. Specified notations to be used as and when required.
4. The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Example of a Use Case Diagram

A use case diagram depicting the Online Shopping website is given below.

Here the Web Customer actor makes use of any online shopping website to purchase online. The top-level uses are as follows; View Items, Make Purchase, Checkout, Client Register. The **View Items** use case is utilized by the customer who searches and view products. The **Client Register** use case allows the customer to register itself with the website for availing gift vouchers, coupons, or getting a private sale invitation. It is to be noted that the **Checkout** is an included use case, which is part of **Making Purchase,** and it is not available by itself.

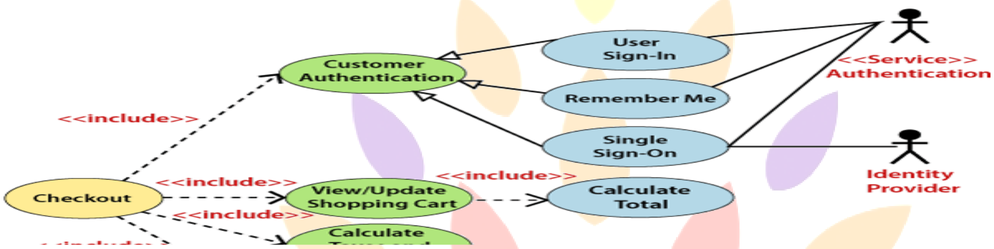


The **View Items** is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item.

Both **View Recommended Item** and **Add to Wish List** include the Customer Authentication use case, as they necessitate authenticated customers, and simultaneously item can be added to the shopping cart without any user authentication.

Similarly, the **Checkout** use case also includes the following use cases, as shown below. It requires an authenticated Web Customer, which can be done by login page, user authentication cookie ("Remember me"), or Single Sign-On (SSO). SSO needs an external identity provider's participation, while Web site authentication service is utilized in all these use cases.

The Checkout use case involves Payment use case that can be done either by the credit card and external credit payment services or with PayPal.



Important tips for drawing a Use Case diagram

Following are some important tips that are to be kept in mind while drawing a use case diagram:

1. A simple and complete use case diagram should be articulated.
2. A use case diagram should represent the most significant interaction among the multiple interactions.
3. At least one module of a system should be represented by the use case diagram.
4. If the use case diagram is large and more complex, then it should be drawn more generalized.

Component diagrams

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

1. Visualize the components of a system.
2. Construct executables by using forward and reverse engineering.
3. Describe the organization and relationships of the components.

How to Draw a Component Diagram?

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

Before drawing a component diagram, the following artifacts are to be identified clearly −

1. Files used in the system.
2. Libraries and other artifacts relevant to the application.
3. Relationships among the artifacts.

After identifying the artifacts, the following points need to be kept in mind.

1. Use a meaningful name to identify the component for which the diagram is to be drawn.
2. Prepare a mental layout before producing the using tools.
3. The following component diagram has been drawn considering all the points

mentioned above.

Where to Use Component Diagrams?

Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to −

4. Model the components of a system.
5. Model the database schema.
6. Model the executables of an application.
7. Model the system's source code.

UNIT-IV TESTING

A strategic Approach for Software testing:

Software Testing is a type of investigation to find out if there is any default or error present in the software so that the errors can be reduced or removed to increase the quality of the software and to check whether it fulfills the specifies requirements or not.

The main objective of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software. The overall strategy for testing software includes:

1. **Before testing starts, it's necessary to identify and specify the requirements of the product in a quantifiable manner.**

Different characteristics quality of the software is there such as maintainability that means the ability to update and modify, the probability that means to find and estimate any risk, and usability that means how it can easily be used by the customers or end-users. All these characteristic qualities should be specified in a particular order to obtain clear test results without any error.

2. Specifying the objectives of testing in a clear and detailed manner.

Several objectives of testing are there such as effectiveness that means how effectively the software can achieve the target, any failure that means inability to fulfill the requirements and perform functions, and the cost of defects or errors that mean the cost required to fix the error. All these objectives should be clearly mentioned in the test plan.

3. For the software, identifying the user's category and developing a profile for each user.

Use cases describe the interactions and communication among different classes of users and the system to achieve the target. So as to identify the actual requirement of the users and then testing the actual use of the product.

4. Developing a test plan to give value and focus on rapid-cycle testing.

Rapid Cycle Testing is a type of test that improves quality by identifying and measuring the any changes that need to be required for improving the process of software. Therefore, a test plan is an important and effective document that helps the tester to perform rapid cycle testing.

5. Robust software is developed that is designed to test itself.

The software should be capable of detecting or identifying different classes of errors. Moreover, software design should allow automated and regression testing which tests the software to find out if there is any adverse or side effect on the features of software due to any change in code or program.

6. Before testing, using effective formal reviews as a filter.

Formal technical reviews is technique to identify the errors that are not discovered yet. The effective technical reviews conducted before testing reduces a significant amount of testing efforts and time duration required for testing software so that the overall development time of software is reduced.

7. Conduct formal technical reviews to evaluate the nature, quality or ability of the test strategy and test cases.

The formal technical review helps in detecting any unfilled gap in the testing approach. Hence, it is necessary to evaluate the ability and quality of the test strategy and test cases by technical reviewers to improve the quality of software.

8. For the testing process, developing a approach for the continuous development.

As a part of a statistical process control approach, a test strategy that is already measured should be used for software testing to measure and control the quality during the development of software.

Testing Strategies for Conventional Software

There are many strategies that can be used to test software.
At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
This approach simply does not work. It will result in buggy software.
At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.
This approach, although less appealing to many, can be very effective.

Types:

1. Unit Testing
2. Integration Testing
3. Validation Testing and
4. System Testing

1. Unit Testing:

Unit testing is a type of software testing where individual units or components of a software are tested. It is concerned with functional correctness of the standalone modules. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

Why Unit Testing?

Unit Testing is important because software developers sometimes try saving time doing minimal unit testing and this is myth because inappropriate unit testing leads to high cost Defect fixing during System Testing, Integration Testing and even Beta Testing after application is built. If proper unit testing is done in early development, then it saves time and money in the end.

Here, are the key reasons to perform unit testing:

1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the code base and enables them to make changes quickly
3. Good unit tests serve as project documentation

4. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code until the tests run again.

5. Integration Testing

Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.

Once all the components or modules are working independently, then we need to check the data flow between the dependent modules is known as **integration testing**.

Types of Integration Testing

Integration testing can be classified into two parts:

1. **Incremental integration testing**
2. **Non-incremental integration testing**

Incremental Approach

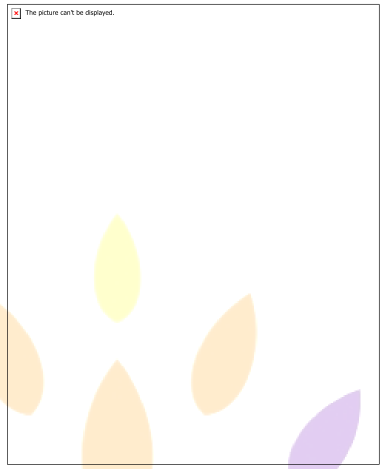
In the Incremental Approach, modules are added in ascending order one by one or according to need. The selected modules must be logically related. Generally, two or more than two modules are added and tested to determine the correctness of functions. The process continues until the successful testing of all the modules.

Incremental integration testing is carried out by further methods:

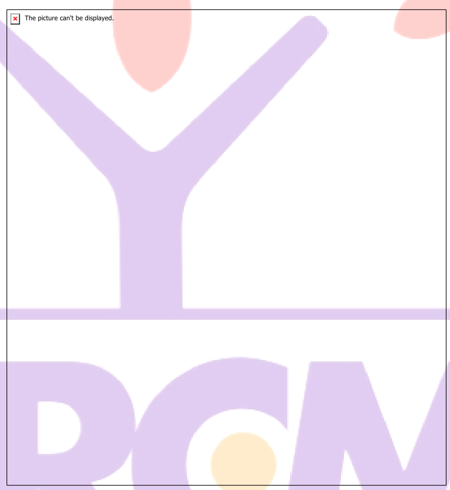
1. Top-Down approach
2. Bottom-Up approach

Top-Down Approach

The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules. Major design flaws can be detected and fixed early because critical modules tested first. In this type of method, we will add the modules incrementally or one by one and check the data flow in the same order.



In the top-down approach, we will be ensuring that the module we are adding is the **child of the previous one like Child C is a child of Child B** and so on as we can see in the below image:



Advantages:

3.Identification of defect is difficult.
4.An early prototype is possible.

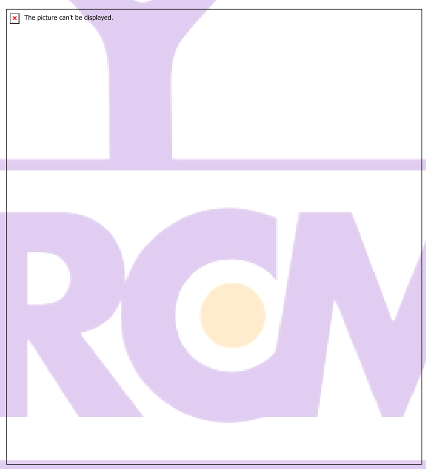
Disadvantages:

5.Due to the high number of stubs, it gets quite complicated.
6.Lower level modules are tested inadequately.
7.Critical Modules are tested first so that fewer chances of defects.

Bottom-Up Method

The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules. Top level critical modules are tested at last, so it may cause a defect. Or we can say that we will be adding the modules from **bottom to the top** and check the data flow in the same order.

In the bottom-up method, we will ensure that the modules we are adding **are the parent of the previous one** as we can see in the below image:



Advantages

8.Identification of defect is easy.
9.Do not need to wait for the development of all the modules as it saves time.

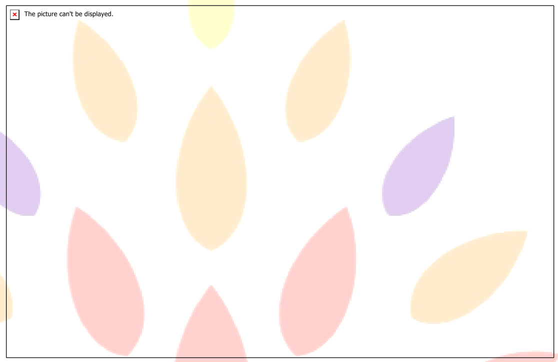
Disadvantages

10. Critical modules are tested last due to which the defects can occur.
11. There is no possibility of an early prototype.

In this, we have one addition approach which is known as **hybrid testing**.

Hybrid Testing Method

In this approach, both **Top-Down** and **Bottom-Up** approaches are combined for testing. In this process, top-level modules are tested with lower level modules and lower level modules tested with high-level modules simultaneously. There is less possibility of occurrence of defect because each module interface is tested.



Advantages

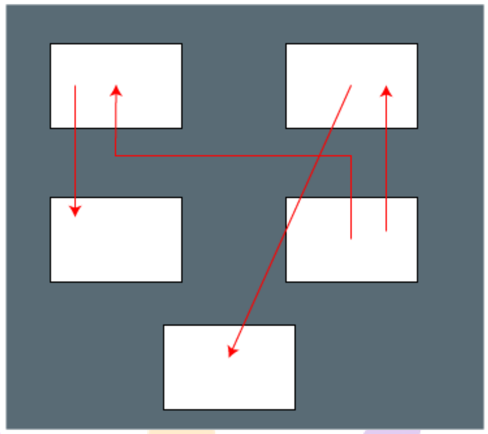
12. The hybrid method provides features of both Bottom Up and Top Down methods.
13. It is most time reducing method.
14. It provides complete testing of all modules.

Disadvantages

15. This method needs a higher level of concentration as the process carried out in both directions simultaneously.
16. Complicated method.

Non- incremental integration testing

We will go for this method, when the data flow is very complex and when it is difficult to find who is a parent and who is a child. And in such case, we will create the data in any module bang on all other existing modules and check if the data is present. Hence, it is also known as the **Big bang method**.



6. Validation Testing

Verification and Validation Testing

Verification testing

Verification testing includes different activities such as business requirements, system requirements, design review, and code walkthrough while developing a product.

It is also known as static testing, where we are ensuring that "**we are developing the right product or not**". And it also checks that the developed application fulfilling all the requirements given by the client.

Validation testing

Validation testing is testing where tester performed functional and non-functional testing. Here **functional testing** includes Unit Testing (UT), Integration Testing (IT) and System Testing (ST), and **non-functional** testing includes User acceptance testing (UAT).

Validation testing is also known as dynamic testing, where we are ensuring that **"we have developed the product right."** And it also checks that the software meets the business needs of the client.

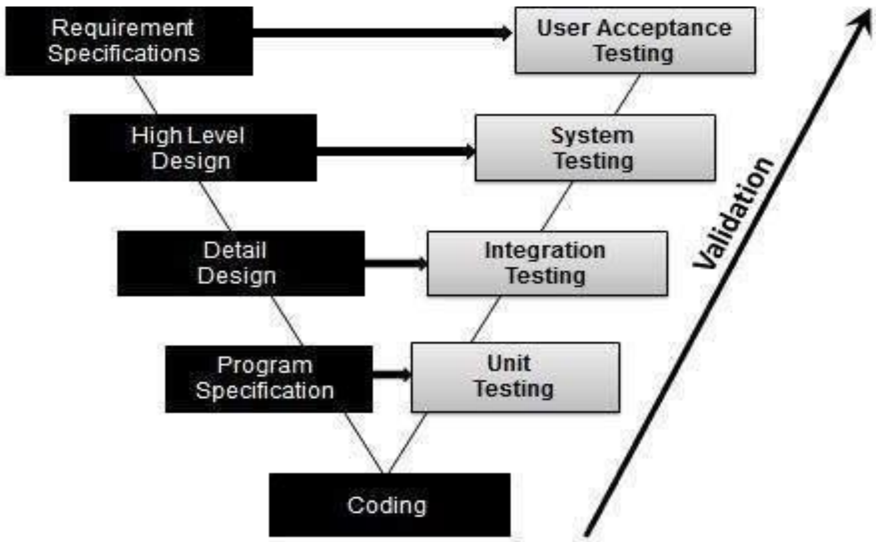
The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

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.

It answers to the question, Are we building the right product?

Validation Testing - Workflow:

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



7. System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system). The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware. System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.

To check the end-to-end flow of an application or the software as a user is known as **System testing**. In this, we navigate (go through) all the necessary modules of an application and check if the end features or the end business works fine, and test the product as a whole system.

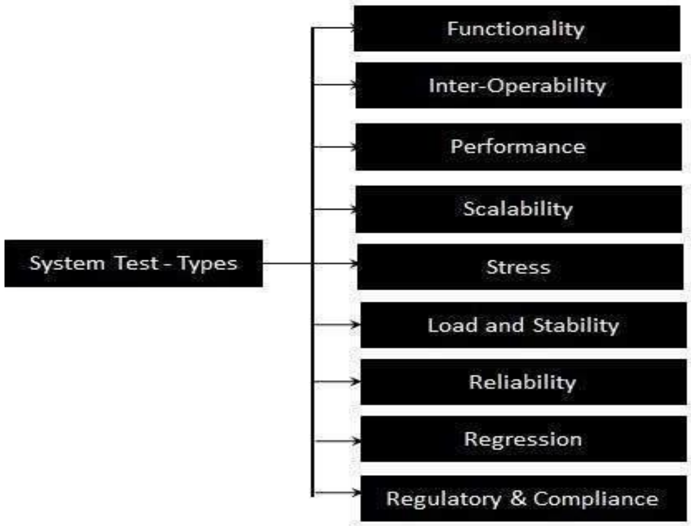
It is **end-to-end testing** where the testing environment is similar to the production environment.

System Testing includes the following steps.

1. Verification of input functions of the application to test whether it is producing the expected output or not.

2. Testing of integrated software by including external peripherals to check the interaction of various components with each other.
3. Testing of the whole system for End to End testing.
4. Behavior testing of the application via a user's experience

Types of System Tests:



Software Testing

1. Two major categories of software testing
1. Black box testing
2. White box testing

Black box testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

How to do Black Box Testing?

Here are the generic steps followed to carry out any type of Black Box Testing.

1. Initially, the requirements and specifications of the system are examined.
2. Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
3. Tester determines expected outputs for all those inputs.
4. Software tester constructs test cases with the selected inputs.
5. The test cases are executed.

6. Software tester compares the actual outputs with the expected outputs.
7. Defects if any are fixed and re-tested.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones -

Functional testing - This black box testing type is related to the functional requirements of a system; it is done by software testers.

Non-functional testing - This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.

Regression testing - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

Equivalence Class Partitioning: It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.

Boundary Value Analysis: Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.

Decision Table Testing: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Equivalence Partitioning Testing

Equivalence Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc. also called as equivalence class partitioning. It is abbreviated as ECP. It is a software testing technique that divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.

An advantage of this approach is it reduces the time required for performing testing of a software due to less number of test cases.

Example:

The Below example best describes the equivalence class Partitioning:

Assume that the application accepts an integer in the range 100 to 999 Valid Equivalence Class partition: 100 to 999 inclusive.

Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters.

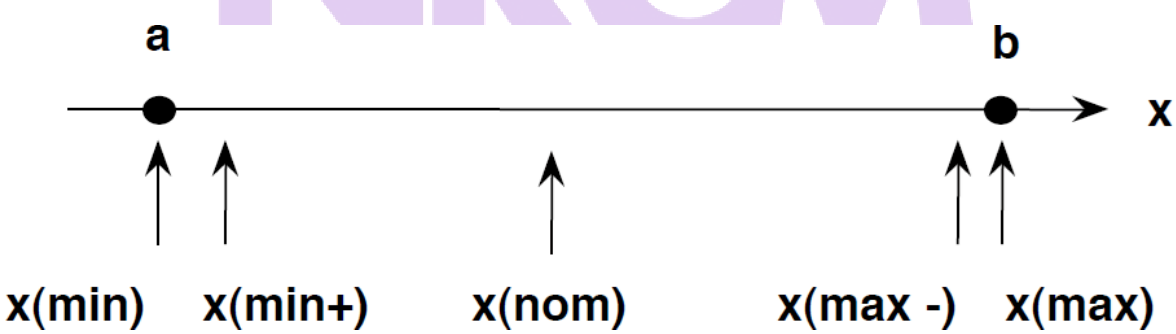
Boundary Value Analysis

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

1. So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".

2. The basic idea in boundary value testing is to select input variable values at their:

1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum



Example: Input Box should accept the Number 1 to 10

Here we will see the Boundary Value Test Cases

Test Scenario Description	Expected Outcome
Boundary Value = 0	System should NOT accept
Boundary Value = 1	System should accept
Boundary Value = 2	System should accept

Interpretation:

1. Case 1 – Username and password both were wrong. The user is shown an error message.
2. Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
3. Case 3 – Username was wrong, but the password was correct. The user is shown an error message.
4. Case 4 – Username and password both were correct, and the user navigated to homepage While converting this to test case, we can create 2 scenarios,
5. Enter correct username and correct password and click on login, and the expected result will be the user should be navigated to homepage

And one from the below scenario

6. Enter wrong username and wrong password and click on login, and the expected result will be the user should get an error message
7. Enter correct username and wrong password and click on login, and the expected result will be the user should get an error message
8. Enter wrong username and correct password and click on login, and the expected result will be the user should get an error message

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.

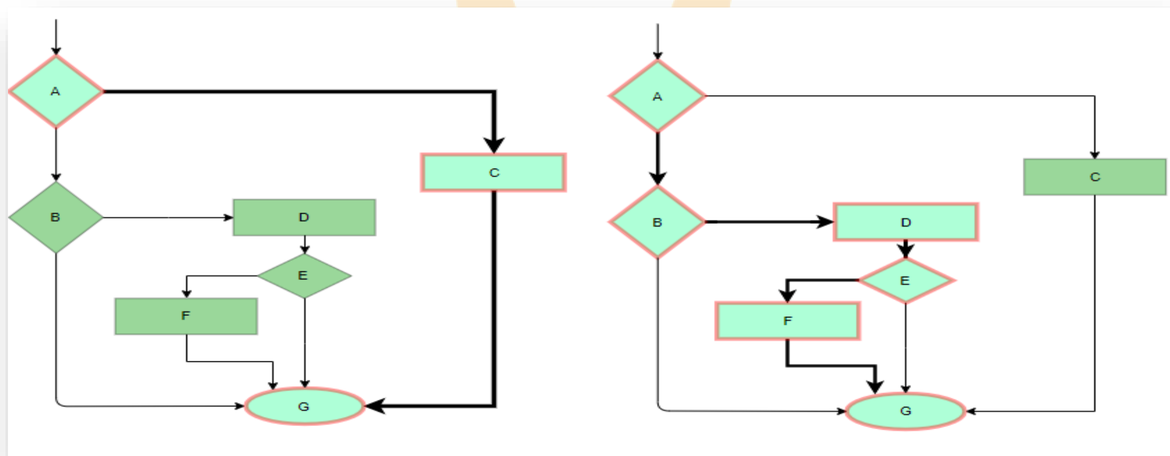
White Box Testing Techniques:

9. **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
10. **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.

11. **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered.

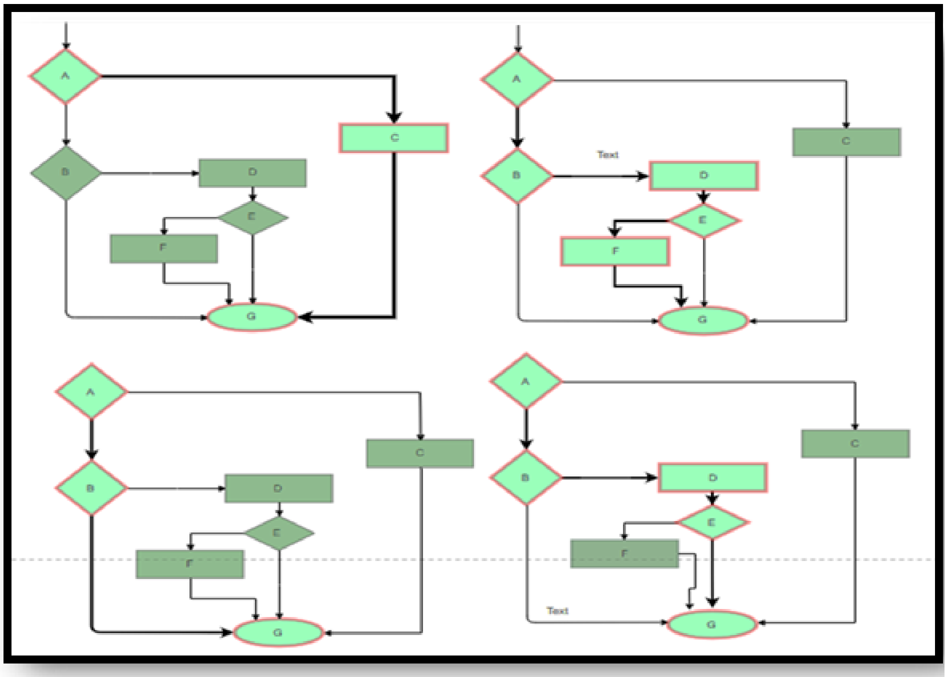
Statement coverage:

In this technique, the aim is to traverse all statement at least once. Hence, each line of code is tested. In case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



Statement Coverage Example

Branch Coverage: In this technique, test cases are designed so that each branch from all decision points are traversed at least once. In a flowchart, all edges must be traversed at least once.



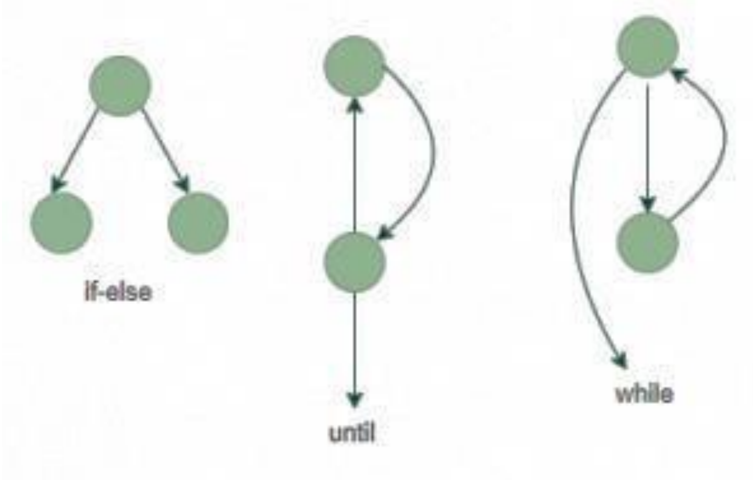
4 test cases required such that all branches of all decisions are covered, i.e, all edges of flowchart are covered

Basis Path Testing: 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.

Steps:

1. Make the corresponding control flow graph
2. Calculate the cyclomatic complexity
3. Find the independent paths
4. Design test cases corresponding to each independent path

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 thatcontains a condition after which the graph splits. Regions are bounded by nodes and edges.

Metrics for Process and Products

Software Measurement: A measurement is a manifestation of the size, quantity, amount or dimension of a particular attributes of a product or process.

It is an authority within software engineering. Software measurement process is defined and governed by ISO Standard.

Need of Software Measurement:

Software is measured to:

1. Create the quality of the current product or process.
2. Anticipate future qualities of the product or process.
3. Enhance the quality of a product or process.
4. Regulate the state of the project in relation to budget and schedule.

Classification of Software Measurement:

There are 2 types of software measurement:

1. Direct Measurement:
In direct measurement the product, process or thing is measured directly using standard scale.

2. Indirect Measurement:
In indirect measurement the quantity or quality to be measured is measured using related parameter i.e. by use of reference.

Metrics:

A metrics is a measurement of the level that any impute belongs to a system product or process. There are 4 functions related to software metrics:

1. Planning
2. Organizing
3. Controlling
4. Improving

Characteristics of software Metrics:

1. Quantitative:
Metrics must possess quantitative nature. It means metrics can be expressed in values.

2. Understandable:
Metric computation should be easily understood, the method of computing metric should be clearly defined.

3. Applicability:
Metrics should be applicable in the initial phases of development of the software.

4. Repeatable:
The metric values should be same when measured repeatedly and consistent in nature.

5. Economical:
Computation of metric should be economical.

6. Language Independent:
Metrics should not depend on any programming language.

Metrics for software quality:

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.

Software quality metrics can be further divided into three categories −

1. Product quality metrics
2. In-process quality metrics
3. Maintenance quality metrics

Product Quality Metrics

This metrics include the following −

4.Mean Time to Failure
5.Defect Density
6.Customer Problems
7.Customer Satisfaction

Mean Time to Failure

It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

Defect Density

It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

Customer Problems

It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.

The problems metric is usually expressed in terms of **Problems per User-Month (PUM)**. PUM = Total Problems that customers reported (true defect and non-defect oriented problems) for a time period + Total number of license months of the software during

the period Where,

Number of license-month of the software = Number of install license of the software × Number of months in the calculation period

PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year.

Customer Satisfaction

Customer satisfaction is often measured by customer survey data through the five-point scale −

8. Very satisfied
9. Satisfied
10. Neutral
11. Dissatisfied
12. Very dissatisfied

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example −

13. Percent of completely satisfied customers
14. Percent of satisfied customers
15. Percent of dis-satisfied customers
16. Percent of non-satisfied customers Usually, this percent satisfaction is used.

In-process Quality Metrics

In-process quality metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes −

17. Defect density during machine testing
18. Defect arrival pattern during machine testing
19. Phase-based defect removal pattern
20. Defect removal effectiveness

Defect density during machine testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort.

This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect arrival pattern during machine testing

The overall defect density during testing will provide only the summary of the defects. The pattern of defect arrivals gives more information about different quality levels in the field. It includes the following −

21. The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.
22. The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.
23. The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

Phase-based defect removal pattern

This is an extension of the defect density metric during testing. In addition to testing, it tracks the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.

Because a large percentage of programming defects is related to design problems, conducting formal reviews, or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management. **Defect removal effectiveness**

It can be defined as follows –

DRE=Defect removed during a development phase X 100%

Defects latent in the product

This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called **early defect removal** when used for the front-end and **phase effectiveness** for specific phases. The higher the value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.

Maintenance Quality Metrics

Although much cannot be done to alter the quality of the product during this phase, following are the fixes that can be carried out to eliminate the defects as soon as possible with excellent fix quality.

24. Fix backlog and backlog management index
25. Fix response time and fix responsiveness
26. Percent delinquent fixes
27. Fix quality

Fix backlog and backlog management index

Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.

Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.

$$BMI = \frac{Number\ of\ problems\ closed\ during\ the\ month}{Number\ of\ problems\ arrived\ during\ the\ month} \times 100\%$$

If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

Fix response time and fix responsiveness

The fix response time metric is usually calculated as the mean time of all problems from open to close. Short fix response time leads to customer satisfaction.

The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer.

Percent delinquent fixes

It is calculated as follows −

$$Percent\ Delinquent\ Fixes = \frac{Number\ of\ fixes\ that\ exceeded\ the\ response\ time\ criteria\ by\ ceverity\ level}{Number\ of\ fixes\ delivered\ in\ a\ specified\ time} \times 100\%$$

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.

A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure; the second is a process measure. The difference between the two dates is the latent period of the defective fix.

Usually the longer the latency, the more will be the customers that get affected. If the number of defects is large, then the small value of the percentage metric will show an optimistic picture. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.

UNIT-V RISK MANAGEMENT

1. 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.*

1. project team reacts to risks when they occur
2. mitigation—plan for additional resources in anticipation of fire fighting
3. fix on failure—resource are found and applied when the risk strikes
4. 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.

1. formal risk analysis is performed
2. organization corrects the root causes of risk
    1. examining risk sources that lie beyond the bounds of the software
    2. 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 tosell,
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.

6. 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:

1. ***Product size***—risks associated with the overall size of the software to be built or modified.
2.
3. ***Business impact***—risks associated with constraints imposed by management or the marketplace.
4. ***Customer characteristics***—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
5. ***Process definition***—risks associated with the degree to which the software process has been defined and is followed by the development organization.
6. ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.
7. ***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.
8. ***Staff size and experience***—risks associated with the overall technical and project experience of the software engineers who will do the work.

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 theircustomers?
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?

3.2 Risk Components and Drivers

The risk components are defined in the following manner:

1. ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
2. ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
3. ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
4. ***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.

5. 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.

Developing a Risk Table Building a Risk

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

Risk	Probability	Impact	RMMM
			Risk Mitigation Monitoring & Management

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.

Assessing Risk Impact

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

1. The ***nature*** of the risk indicates the problems that are likely if it occurs.
2. The ***scope*** of a risk combines the severity (just how serious is it?) with its overalldistribution.
3. 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.

4. 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.

5. RISK MITIGATION, MONITORING, AND MANAGEMENT

An effective strategy must consider three issues:

1. Risk avoidance
2. Risk monitoring
3. 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

4. Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, and competitive job market).
5. Mitigate those causes that are under our control before the project starts.
6. Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
7. Organize project teams so that information about each development activity is widely dispersed.
8. Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
9. 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:

10. General attitude of team members based on project pressures.
11. The degree to which the team has jelled.
12. Interpersonal relationships among team members.
13. Potential problems with compensation and benefits
14. 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.

15. 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 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

1. 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 softwareprocess,
4. a multitiered testing strategy,
5. control of software documentation and the changes made to it,

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

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.

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.

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

1. quality planning
2. formal technical reviews
3. test equipment
4. training

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

5. in-process and interprocess inspection
6. equipment calibration and maintenance
7. 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

8. rework
9. repair
10. 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

11. complaint resolution
12. product return and replacement
13. help line support
14. warranty work

6.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 surely result.
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.

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

SQA Activities

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

1. the software engineers who do technical work and

2. 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 byall interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

3. evaluations to be performed
4. audits and reviews to be performed
5. standards that are applicable to the project
6. procedures for error reporting and tracking
7. documents to be produced by the SQA group
8. 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.

9. 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.

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

1. during design will cost 1.0 monetary unit to correct.
2. just before testing commences will cost 6.5 units;
3. during testing, 15 units;

4. and after release, between 60 and 100 units.

3.2) Defect Amplification and Removal:

(This topic I will tell you later)

5. 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.

The Review Meeting

Every review meeting should abide by the following constraints:

1. Between three and five people (typically) should be involved in the review.
2. Advance preparation should occur but should require no more than two hours of work for each person.
3. 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.

1. 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.
2. Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
3. 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.

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.

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 isrecorded.
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.

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…

1. Inspect a fraction a_i of each software work product, *i*. Record the number of faults, f_i found within a_i.

2. Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by$1/a_i$.
3. Sort the work products in descending order according to the gross estimate of the number of faults in each.
4. 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

1. Be representative of the work product as a whole and

Large enough to be meaningful to the reviewer(s) who does the sampling

2. 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 For software, statistical quality assurance implies the following steps:

.

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*.

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:

1. ***Define*** customer requirements and deliverables and project goals via well-defined methods of customer communication
2. ***Measure*** the existing process and its output to determine current quality performance (collect defect metrics)
3. ***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.

4. ***Improve*** the process by eliminating the root causes of defects.
5. ***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:

6.***Design*** the process to
1.avoid the root causes of defects and
2.to meet customer requirements
7.***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.

5. 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 beendeveloped to help interpret the standard for use in the software process. The requirements delineated by ISO 9001 address topics such as

1. management responsibility,
2. quality system, contract review,
3. design control,
4. document and data control,
5. product identification and traceability,
6. process control,
7. inspection and testing,
8. corrective and preventive action,
9. control of quality records,
10. internal quality audits,
11. training,
12. servicing and

13. 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.

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".

7.1 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.

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

1. causes uncontrolled acceleration that cannot be stopped
2. does not respond to depression of brake pedal (by turning off)
3. does not engage when switch is activated
4. slowly loses or gains speed

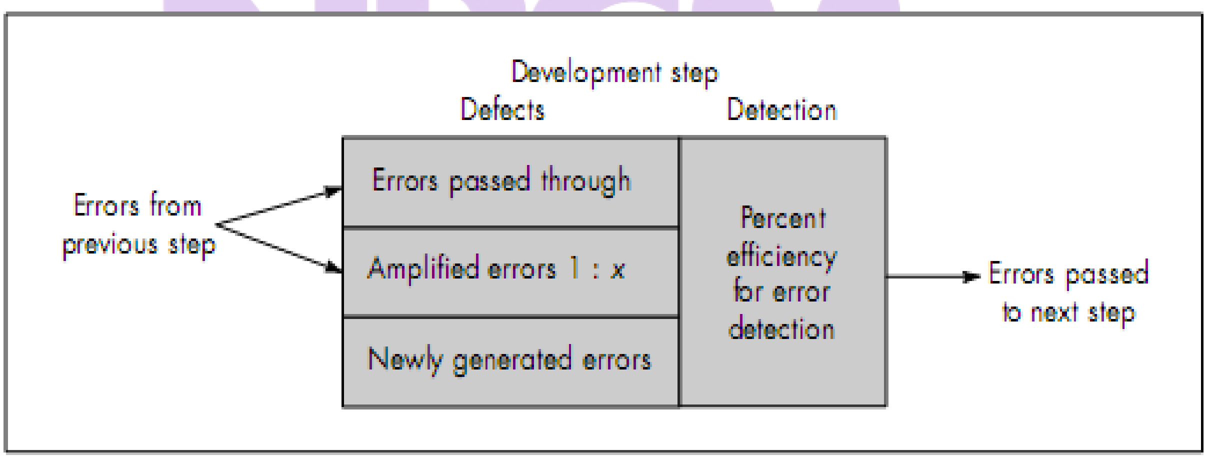
Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.To be effective, software must be analyzed in the context of the entire system.

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.

Defect Amplification and Removal:



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.

Referring to the figure8.3 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.

Figure8.4 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).

1. ***Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units.***
2. ***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.

FIGURE 8.3

Defect amplification, no reviews

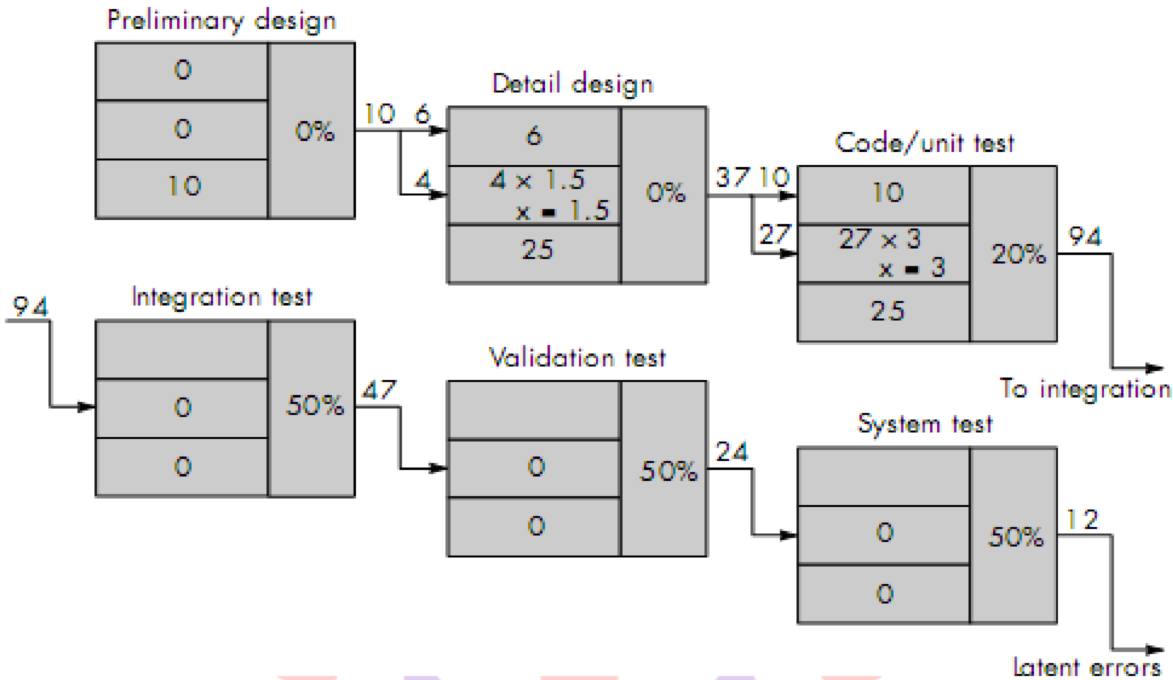
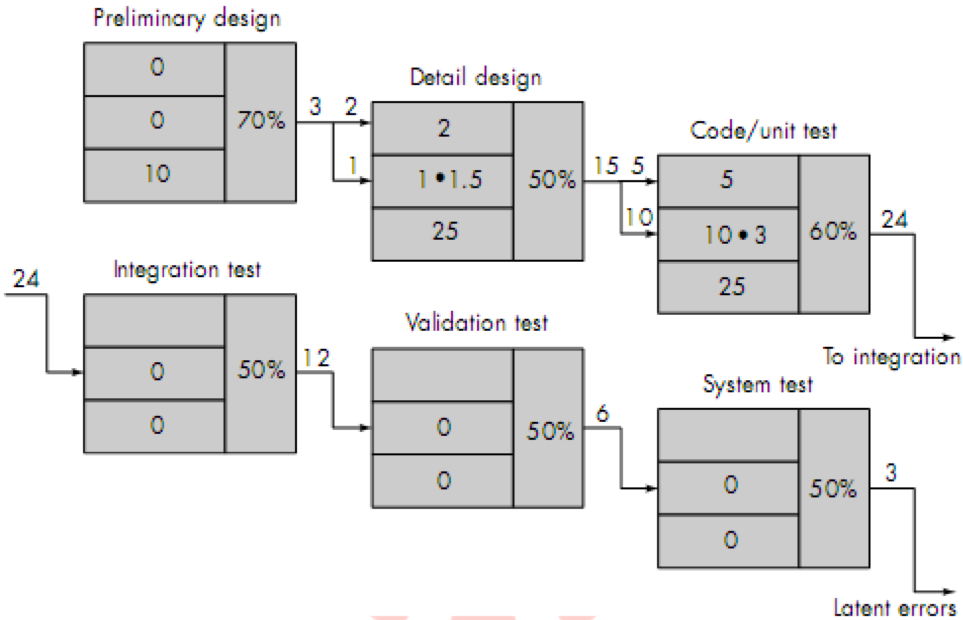


FIGURE 8.4

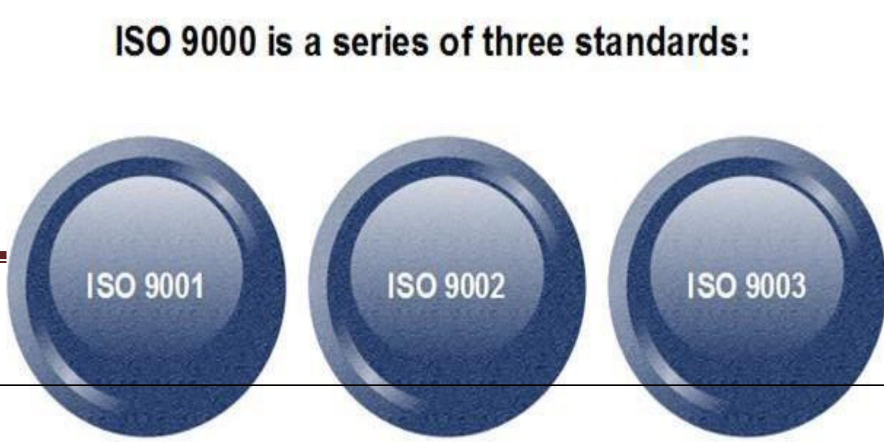
Defect amplification, reviews conducted



ISO 9000 Certification:

ISO (International Standards Organization) is a group or consortium of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

Types of ISO 9000 Quality Standards



The ISO 9000 series of standards is based on the assumption that if a proper stage is followed for production, then good quality products are bound to follow automatically. The types of industries to which the various ISO standards apply are as follows.

1. **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

How to get ISO 9000 Certification?

An organization determines to obtain ISO 9000 certification applies to ISO registrar office for registration. The process consists of the following stages:

1. **Application:** Once an organization decided to go for ISO certification, it applies to the registrar for registration.
2. **Pre-Assessment:** During this stage, the registrar makes a rough assessment of the organization.
3. **Document review and Adequacy of Audit:** During this stage, the registrar reviews the document submitted by the organization and suggest an improvement.
4. **Compliance Audit:** During this stage, the registrar checks whether the organization has compiled the suggestion made by it during the review or not.
5. **Registration:** The Registrar awards the ISO certification after the successful completion of all the phases.
6. **Continued Inspection:** The registrar continued to monitor the organization time by time.