# MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

***COMPUTER SCIENCE AND ENGINEERING***

## COURSE FILE

**Subject: SOFTWARE ENGINEERING**

**Class: III/ IV B. Tech (CSE) I-SEM**

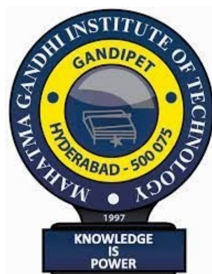

**Prepared By**

**Dr. D. Koteswara Rao**

**Ms. Gousiya Begum**

**Ms. K.Sunitha**

**Faculty of CSE Dept**


**ACADEMIC YEAR 2021-2022**

# INDEX


| S.No | Name of the Experiment | Page No. |
|---|---|---|
| 1 | JNTUH Syllabus | 1 |
| 2 | Course Plan and Lesson Plan | 2 |
| 3 | CO PO PSO Mapping | 5 |
| 4 | UNIT-1 | 6 |
| 5 | UNIT-2 | 17 |
| 6 | UNIT-3 | 29 |
| 7 | UNIT-4 | 50 |
| 8 | UNIT-5 | 65 |
| 9 | Questions from University Papers | 71 |
| 10 | Assignment and MID Term Exam Papers | 77 |
| 11 | Recent Process Models ( Content beyond syllabus) | 80 |
| 12 | Question Bank | 82 |

## CS502PC: SOFTWARE ENGINEERING

**III Year B.Tech. CSE I-Sem**

| L | T | P | C |
|---|---|---|---|
| 3 | 0 | 0 | 3 |

**Course Objectives**

1. The aim of the course is to provide an understanding of the working knowledge of the techniques for estimation, design, testing and quality management of large software development projects.
2. Topics include process models, software requirements, software design, software testing, software process/product metrics, risk management, quality management and UML diagrams

**Course Outcomes**

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 patterns, process assessment, personal and team process models.

**Process models:** The waterfall model, incremental process models, evolutionary process models, the unified process.

**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.

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

**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.

**Product metrics:** Software quality, metrics for analysis model, metrics for design model, metrics for source code, metrics for testing, metrics for maintenance.

**UNIT - V**

**Metrics for Process and Products:** Software measurement, metrics for software quality.

**Risk management:** Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM, RMMM plan.

**MAHATMA GANDHI INSTITUTE OF TECHNOLOGY, GANDIPET, HYDERABAD – 75.**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# COURSE PLAN

B. Tech.(CSE-3) III Year I – Semester : Academic Year : 2021 – 2022
Name of the faculty : D. KOTESWARA RAO
Name of the subject : SOFTWARE ENGINEERING

| S. No | Unit | Brief details of Unit | No. of periods required | Remarks |
|---|---|---|---|---|
| 1 | 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 patterns, process assessment, personal and team process models. Process models: The waterfall model, incremental process models, Evolutionary process models, the unified process. | 12 | - |
| 2 | 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. System models: Context models, behavioral models, data models, object models, structured methods. | 08 | - |
| 3 | 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. | 10 | - |
| 4 | 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. Product metrics: Software quality, metrics for analysis model, metrics for design model, metrics for source code, metrics for testing, metrics for maintenance. | 8 | - |
| 5 | V | Risk management: Reactive vs Proactive Risk strategies, software risks, Risk identification,Risk projection, Risk refinement, RMMM, RMMM Plan. Quality Management: Quality concepts, Software quality assurance, Software Reviews, Formal technical reviews, Statistical Software quality Assurance, Software reliability, The ISO 9000 quality standards. | 7 | - |
| | | **TOTAL PERIODS:** | **45** | |

**Signature of the Faculty** **Signature of the HoD** **Signature of the Principal**

**MAHATMA GANDHI INSTITUTE OF TECHNOLOGY, GANDIPET, HYDERABAD – 75.**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# LESSON PLAN

B. Tech.(CSE-3) III Year I – Semester : Academic Year : 2021 – 2022
Name of the faculty : D. KOTESWARA RAO
Name of the subject : SOFTWARE ENGINEERING

| S. No | Unit No. | Date | Topic | No. of Periods | Cumulative Periods |
|---|---|---|---|---|---|
| 1. | I | 06/09/2021 | Introduction to Software Engineering | 1 | 1 |
| 1. | | 08/09/2021 | The evolving role of software ,Changing Nature of Software | 1 | 2 |
| 2. | | 13/09/2021 | Software myths | 1 | 3 |
| 3. | | 15/09/2021 | A Generic view of process: Software engineering- a layered technology, a process framework | 1 | 4 |
| 4. | | 17/09/2021 | the capability maturity model integration (CMMI), process patterns | 1 | 5 |
| 5. | | 20/09/2021 | Process models: The waterfall model | 1 | 6 |
| 6. | | 22/09/2021 & 24/09/2021 | incremental process models | 2 | 8 |
| 7. | | 27/09/2021 & 29/09/2021 | Evolutionary process models | 2 | 10 |
| 8. | | 01/10/2021 | the unified process | 1 | 11 |
| 9. | | 04/10/2021 | Process assessment, Personal and team process models. | 1 | 12 |
| 10. | II | 08/10/2021 | Introduction to Requirements engineering, Functional requirements | 1 | 13 |
| 11. | | 20/10/2021 | Non-functional requirements, Interface specification | 1 | 14 |
| 12. | | 22/10/2021 | user requirements, system requirements | 1 | 15 |
| 13. | | 25/10/2021 | The software requirements document, Requirements engineering process: Feasibility studies | 1 | 16 |
| 14. | | 27/10/2021 | Requirements elicitation and analysis | 1 | 17 |
| 15. | | 29/10/2021 | Requirements validation, requirements management | 1 | 18 |
| 16. | | 01/11/2021 | System models: Context models, behavioral models | 1 | 19 |
| 17. | | 03/11/2021 | Data models, object models, structured methods. | 1 | 20 |
| 18. | III | 05/11/2021 | Design Engineering: Design process and design quality | 1 | 21 |
| 19. | | 08/11/2021 to 13/11/2021 | **I - MID Exam** | | |
| 20. | | 15/11/2021 | Design concepts | 1 | 22 |
| 21. | | 17/11/2021 | The design model | 1 | 23 |
| 22. | | 19/11/2021 | Creating an architectural design: software architecture | 1 | 24 |
| 23. | | 22/11/2021 | Architectural styles and patterns | 1 | 25 |
| 24. | | 24/11/2021 | Architectural design | 1 | 26 |
| 25. | | 26/11/2021 | Conceptual model of UML | 1 | 27 |
| 26. | | 29/11/2021 | Basic structural modeling | 1 | 28 |
| 27. | | 01/12/2021 | Class diagrams, sequence diagrams | 1 | 29 |

| S. No | Unit No. | Date | Topic | No. of Periods | Cumulative Periods |
|---|---|---|---|---|---|
| 28. | | 03/12/2021 | Collaboration diagrams, use case diagrams, component diagrams | 1 | 30 |
| 29. | IV | 06/12/2021 | Testing Strategies: A strategic approach to software testing, test strategies for conventional software | 1 | 31 |
| 30. | | 08/12/2021 | Test strategies for conventional software | 1 | 32 |
| 31. | | 10/12/2021 | black-box testing | 1 | 33 |
| 32. | | 13/12/2021 | white-box testing | 1 | 34 |
| 33. | | 15/12/2021 | validation testing, system testing, the art of debugging | 1 | 35 |
| 34. | | 17/12/2021 | Product metrics: Software quality, metrics for analysis model | 1 | 36 |
| 35. | | 20/12/2021 | metrics for design model, metrics for source code | 1 | 37 |
| 36. | | 22/12/2021 | Metrics for testing, metrics for maintenance. | 1 | 38 |
| 37. | V | 24/12/2021 | Risk management: Reactive vs Proactive Risk strategies | 1 | 39 |
| 38. | | 27/12/2021 | Software risks, Risk identification, Risk projection | 1 | 40 |
| 39. | | 29/12/2021 | Risk refinement, RMMM, RMMM Plan | 1 | 41 |
| 40. | | 31/12/2021 | Quality Management: Quality concepts, Software quality assurance | 1 | 42 |
| 41. | | 03/01/2022 | Software Reviews, Formal technical reviews | 1 | 43 |
| 42. | | 05/01/2022 | Statistical Software quality Assurance, Software reliability, The ISO 9000 quality standards | 1 | 44 |
| 43. | | 07/01/2022 | Discussion of Previous year question papers | 1 | 45 |

Note: Dussehra Vacation: From 11th Oct. 2021 to 16th Oct. 2021

**Signature of the Faculty** **Signature of the HoD** **Signature of the Principal**

# CO-PO MAPPING

**Subject Name: Software Engineering (R18)** **Year: III**
**Subject Code: CS502PC** **Semester: I**

| | |
|---|---|
| CS502PC. 1 | Ability to apply different Process Models. |
| CS502PC. 2 | Ability to identify minimum requirements for the development of application. |
| CS502PC. 3 | Ability to translate requirements to high level design models. |
| CS502PC. 4 | Ability to conduct appropriate testing strategies and methods. |
| CS502PC. 5 | Ability to measure software using metrics and conduct quality tasks. |

**Subject Name: CS502PC-Software Engineering**

| **CO'S** | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 | PS1 | PS2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CO1 | 3 | 2 | - | - | - | - | - | - | 3 | 2 | 2 | - | | |
| CO2 | 2 | 3 | - | - | - | - | - | 2 | 3 | 3 | - | - | 2 | 3 |
| CO3 | 2 | 2 | - | - | 3 | - | - | - | 2 | 3 | - | - | 2 | 3 |
| CO4 | 2 | 2 | - | - | 3 | - | - | 2 | 3 | 2 | 2 | - | 2 | 3 |
| CO5 | 2 | 2 | - | - | - | - | - | - | 1 | 2 | 3 | - | | |
| Average | 2.2 | 2.2 | - | - | 1.2 | - | - | 0.8 | 2.4 | 2.4 | 1.4 | - | 1.2 | 1.8 |

**1. Software characteristics:**

*1. Software is developed or engineered; it is not manufactured in the classical sense.*

In both software development and hardware manufacturing activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

*2. Software doesn't "wear out."*

Figure 1, depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life due to manufacturing defects; defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, temperature extremes, etc. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. Software doesn't wear out. But it does deteriorate! During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve". Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

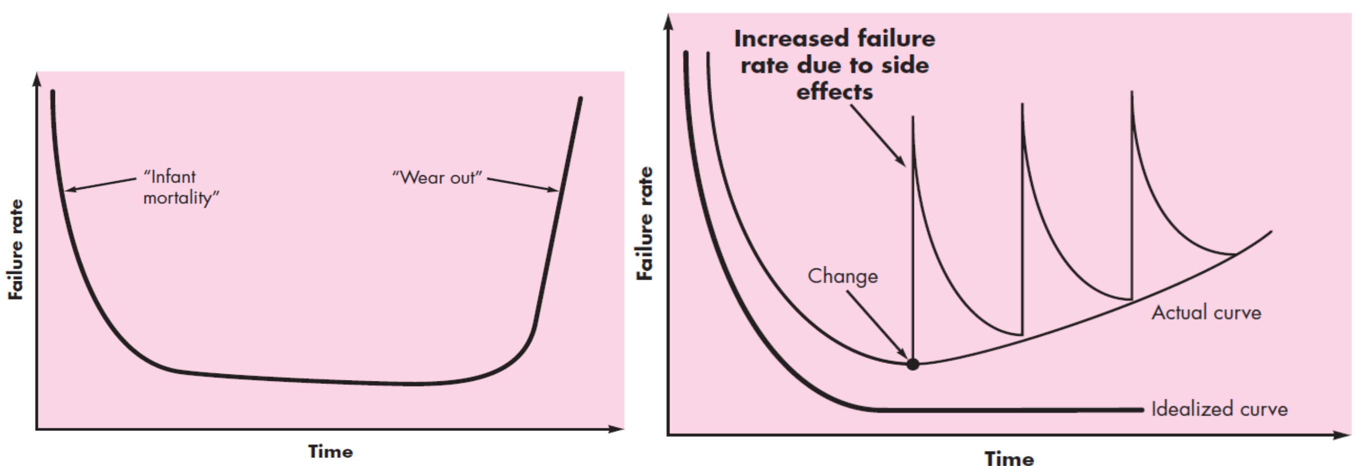



Fig:1 depicts failure rate as a function of time for hardware. Fig:2 failure rate curve for software.

*3. Although the industry is moving toward component-based construction, most software continues to be custom built.*

Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

**2. Software Application Domains:**

Today, seven broad categories of computer software present continuing challenges for software engineers:

*System software*—a collection of programs written to service other programs. Some system software (e.g., compilers, and editors) processes complex, but determinate, information structures. Other systems applications (e.g., operating system components, networking software) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware;

*Application software*—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. Application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

*Engineering/scientific software*—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

*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. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven).

*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 marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management).

*Web applications*—called "WebApps," in their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

*Artificial intelligence software*—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

**3. Software myths-** erroneous beliefs about software and the process that is used to build it. They appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score." Today, most knowledgeable software engineering professionals recognize myths for what they are- misleading attitudes that have caused serious problems for managers and practitioners alike.

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

*Myth:* We already have a book that's full of standards and procedures for building software. Won't 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? In many cases, the answer to all of these questions is "no."

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

*Reality:* As new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. 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 build it.

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

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

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

*Reality:* a comprehensive and stable statement of requirements is not always possible.

Unambiguous requirements are developed only through effective and continuous communication.

*Myth:* Software requirements continually change, but change can be easily accommodated because software is flexible.

*Reality:* the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly.

**Practitioner's myths-** myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. Old ways and attitudes die hard.

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

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

*Reality:* One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review.

*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. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

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

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

Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

**Some Short Notes:**

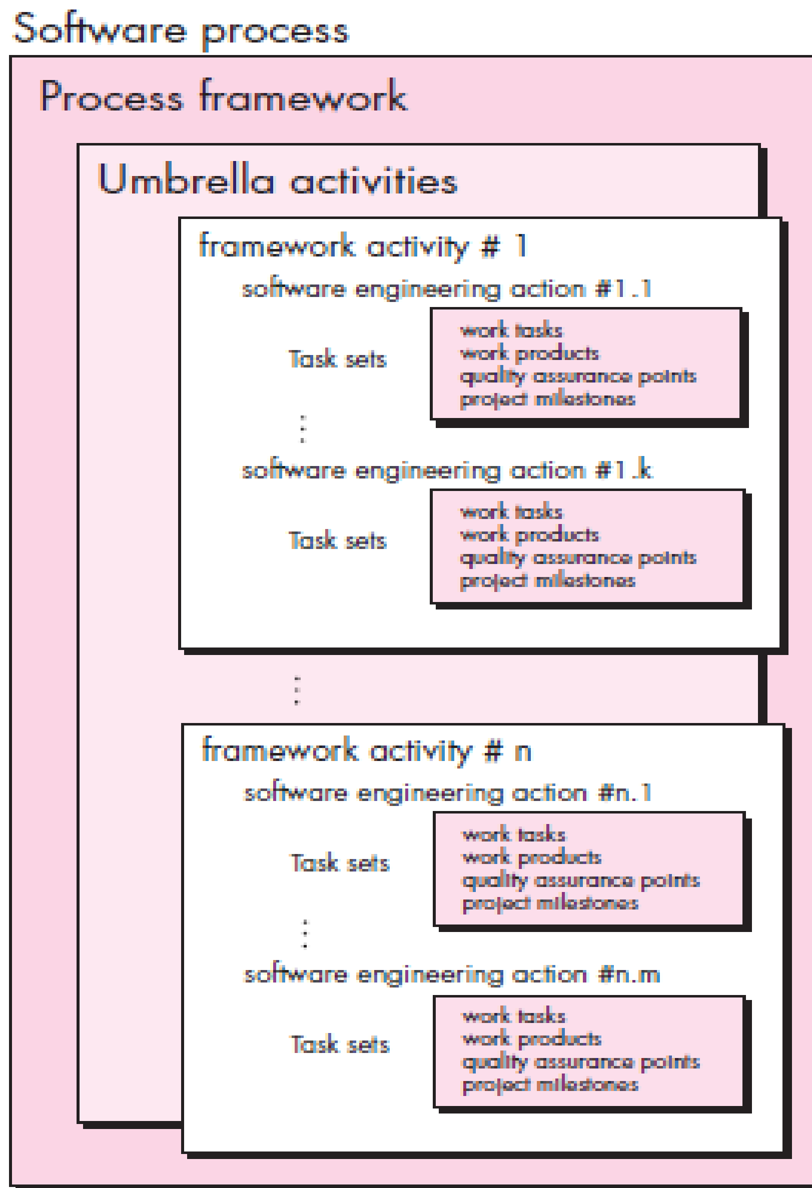
***Legacy software:*** older programs—often referred to as legacy software. This is causing headaches for large organizations who find them costly to maintain and risky to evolve. There is sometimes one additional characteristic that is present in legacy software—poor quality. Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history.

***Role of software*:** software has dual role: product and as the vehicle for delivering a product.

***Software:*** is: (1) instructions that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

***Process flow:*** describes how the framework activities ( communication, planning, modeling, construction, deployment) and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

**4. Process Framework:**




A **software process** is a framework for the activities, actions, and tasks that are required to build high-quality software. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A **generic process framework** for software engineering defines five framework activities—communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

**Process flow**—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time. A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment . An iterative process flow repeats one or more of the activities before

proceeding to the next. An evolutionary process flow executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software. A parallel process flow executes one or more activities in parallel with other activities.

**5. Process Patterns:**

A process pattern provides you with a template —a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project. Patterns can be defined at any level of abstraction. A pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). Patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., planning) or an action within a framework activity (e.g., project estimating).

Ambler [Amb98] has proposed a template for describing a process pattern:

*Pattern Name.* The pattern is given a meaningful name describing it within the context of the software process (e.g., TechnicalReviews).

*Forces.* The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

*Type.* Stage pattern—defines a problem associated with a framework activity for the process. An example of a stage pattern might be EstablishingCommunication. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., RequirementsGathering is a task pattern). Phase pattern—define the sequence of framework activities that occurs within the process. An example of a phase pattern might be SpiralModel.

*Initial context.* Describes the conditions under which the pattern applies. Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists?

*Problem.* The specific problem to be solved by the pattern.

*Solution.* Describes how to implement the pattern successfully. This section describes how the initial state of the process is modified as a consequence of the initiation of the pattern.

*Resulting Context.* Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern: (1) What organizational or team-related activities must have occurred? (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

*Related Patterns.* Provide a list of all process patterns that are directly related to this one.

*Known Uses and Examples.* Indicate the specific instances in which the pattern is applicable. For

example, Communication is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

**6. Process assessment and improvement:**

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice. In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

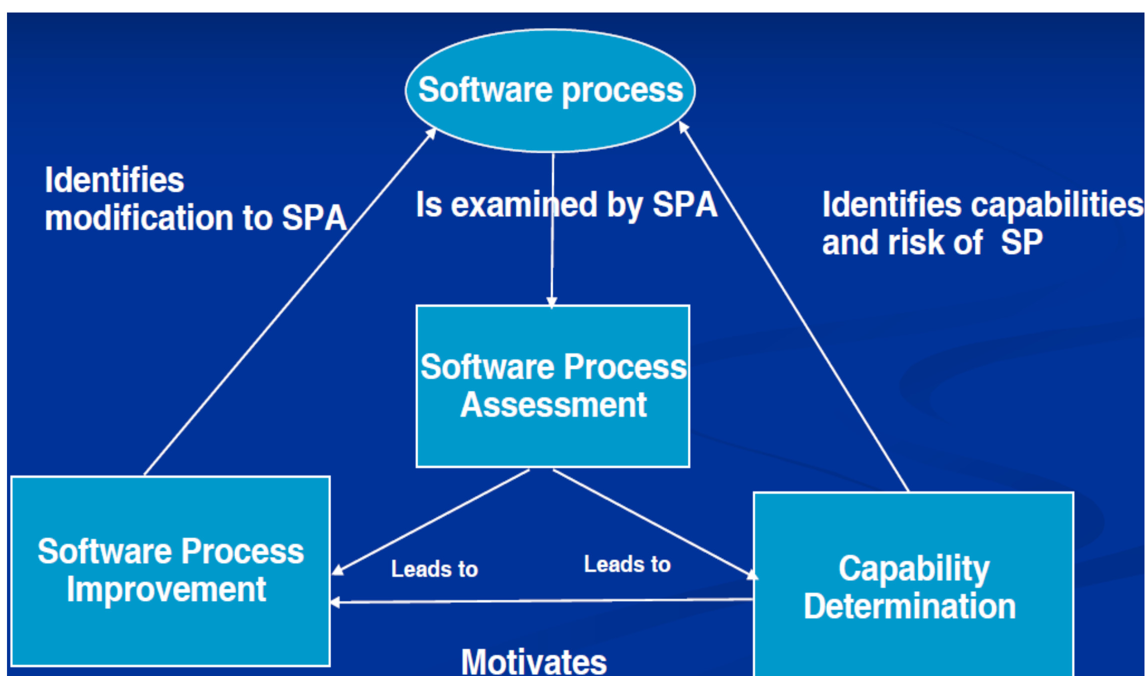
A number of different approaches to software process assessment and improvement have been proposed:

*Standard CMMI Assessment Method for Process Improvement (SCAMPI)*—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning.

*CMM-Based Appraisal for Internal Process Improvement (CBA IPI)*— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.

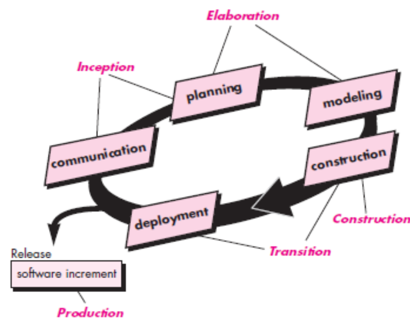
*SPICE (ISO/IEC15504)*—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

*ISO 9001:2000* for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.



**7. Unified process:**

Unified Process, a framework for object-oriented software engineering using unified modeling language (UML). Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds. The iterative, incremental model proposed by the UP can be adapted to meet specific project needs.




The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models. The Unified Process recognizes the importance of customer communication and methods for describing the customer's view of a system (the use case). It emphasizes the important role of software architecture. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

**Phases of the Unified Process:**

*The inception phase* of the UP encompasses both customer communication and planning activities. Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires. A rough architecture for the system is proposed. Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

*The elaboration phase* encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

*The construction phase* develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed. All necessary and required features and functions for the software increment are then implemented in source code. The unit tests are designed and

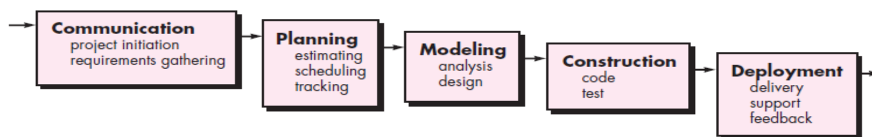
executed for each. In addition, integration activities (component assembly and integration testing) are conducted.

In the *transition phase* of the UP, software is given to end users for beta testing and user feedback reports both defects and necessary changes. The software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.

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

**8. The Waterfall Model:**

It can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner. There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion.



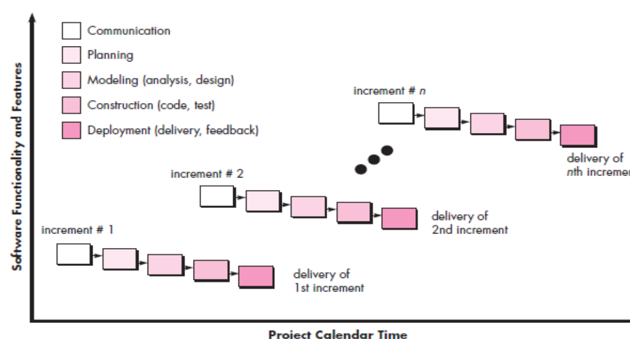

This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made. The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software. Among 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.

2) It is often difficult for the customer to state all requirements explicitly

3) The customer must have patience. A working version of the program(s) will not be available until late in the project time span.

**9. Incremental Process Models:**

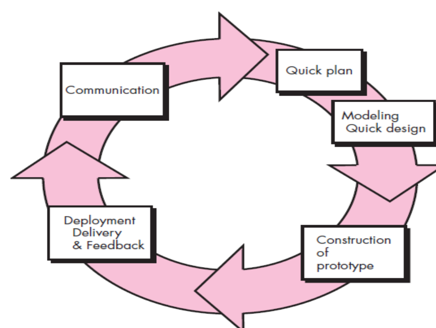


There may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment;

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced. Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

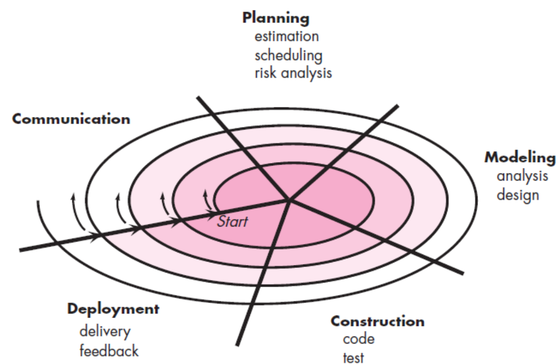
**10. Evolutionary Process Models:**

**Prototyping:** Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In this case, a prototyping paradigm may offer the best approach. The prototype can serve as "the first system." Ideally, the prototype serves as a mechanism for identifying software requirements. The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users. The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.




**The Spiral Model:**

the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.




Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. 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. Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product.

***Short Notes:*** ***Personal software process***- *t*he Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities: planning, high-level design, high-level design review, development, postmortem.

***Team software process (TSP)***- because many industry-grade software projects are addressed by a team of practitioners, the goal of TSP is to build a "selfdirected" project team that organizes itself to produce high-quality software. The following objectives for TSP:Build self-directed teams that plan and track their work, establish goals, and own their processes and plans; show managers how to coach and motivate their teams.

***Process assessment:*** assessing a process to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

***Software process*** : is a framework for the activities, actions, and tasks that are required to build high-quality software (or) a collection of work activities, actions, and tasks that are performed when some work product is to be created.

***Umbrella activities:*** are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

***Software engineering layers:*** tools, methods, process, a quality focus.

**An activity:** strives to achieve a broad objective (e.g., communication with stakeholders). An action (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model). A task focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

**Requirements, Requirements Engg, System Models ( UNIT-II)**

**1. Functional and non-functional requirements:**

Software requirements are often classified as functional and non-functional. *Functional requirements* are statements of services the systems should provide, how the system should react particular inputs, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, these may also explicitly state what the system should not do. These depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements functional requirements usually described in an abstract way that can be understood by system users. The functional user requirements define specific facilities to be provided by the system. Functional system requirements vary from general requirements covering what the system should do to very specific requirements.

***Non-functional requirements:*** These are constraints on the services (or) functions offered by the system. They include timing constraints, constraints on development process, and constraints imposed by standards. These often apply to the system as a whole, rather than individual system features (or) services. These may affect the overall architecture of a system rather than individual components.

***Types of non-functional requirements-***

*Product requirements*- specify or constrain the behavior of the software. Examples: performance-how fast it must execute, how much memory it needs, etc.

*Organisational requirements*- are derived from policies and procedures in the customer's and developer's oranization. Example: how system will be used (operational process requirements)

*External requirements*- are derived from the factors external to the system and development process. Examples: interoperability requirements, legislative requirements, etc.

***Requirements engineering***- the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

**2. User requirements/ System requirements:**

*User requirements*- the high-level abstract statements in a natural language and using diagrams of what services the system is expected to provide to system users and constrainsts under which it must operate. User requirements are quite general.

*System requirements*- are more detailed descriptions of the software system's functions, services and operational constrainsts. The system requirements documents should define exactly what is to be implemented. The system requirements provvide more specific information about the services and functions of the system that is to be implemented.

Example: Mental health care patient management system (MHC-PMS)

User Requirements Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of drugs prescribed, their cost, and prescribing clincs shall be generated.

1.2 The system shall automatically generate the report for printing after 17:30 on the last working day of the month.

1.3 A report shall be created for each clinic and shall list drug names, the total number of prescriptions, the number of doeses and the total cost of the drugs.

1.4 If drugs are available in different dose units, separate reprots shall be created for each dose unit.

1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

**3. Requirements engineering process:**

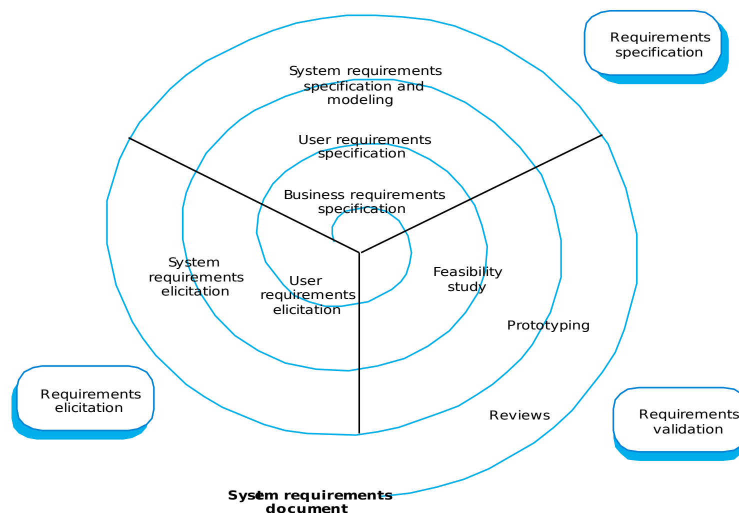



Fig: shows the interleaving of activities to carry out requirements engineering process.

This includes four high-level activities: feasibility *study, elicitation and analysis, specification, and validation*. Feasibility study consists of assessing if the system is useful to the business or not. The purpose of requirements elicitation and analysis is for discovering requirements. Specification means converting the requirements into some standard form. Validation involves checking that the requirements actually define the system that the customer wants.

In practice, requirements engineering is an iterative process in which the activities are interleaved.

In the above diagram, activities are organized as an iterative process around a spiral, with output being a system requirements document. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, most effort will be devoted to eliciting and understanding the detailed system requirements. 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.

**4. Requirements elicitation and analysis:**

After an initial feasibility study, the next stage of requirements engineering process is requirements elicitation and analysis. Here, software engineers work with all stakeholders such as customers and system end-users to find out the application domain, what services the system should provide, the required performance, hardware constraints and so on. A system stakeholder is anyone who should have some direct or indirect influence on the system.

Elicitation and analysis consists of different activities: Requirements discovery, Requirements classification and organization, prioritization and negotiation, and specification.

*Discovery*- it is the process of interacting with stakeholders of the system to discover requirements. Domain requirements from stakeholders and documentations are also discovered during this activity.

*Classification and organization:* It takes unstructured collection of requirements, groups related requirements and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.

*Prioritization and negotiation:* when multiple stakeholders are involved, requirements will conflict. It is concerned with prioritizing requirements, and finding and resolving requirements conflicts through negotiation. Usually stakeholders have to meet to resolve differences and agree an compromise requirements.

*Specification:* requirements are documented and input into the next round of the spiral. Formal and informal documents may be produced.

**5. Validation techniques:**

*Reviews:* the requirements are analyzed systematically by a team of reviews who check for errors and inconsistencies.

*Prototyping:* an executable model of the system that is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

*Test case generation:* requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult, this means requirements will be difficult to implement.

## System Modeling

***System modeling*** is the process of developing abstract models of a system, with each model presenting a different view (or) perspective of that system. System modeling means representing the system using some kind of graphical notation, which is always based on notations in the UML.

***Model*** is an abstraction of the system being studied rather than alternative representation of that system. An ***abstraction*** deliberately simplifies and picks out the most salient characteristics. Ex.

an abstraction of the book’s key points. Models are used during the requirements engineering process to help derive the requirements for a system.

**6. System models-** we may develop different models to represent the system from different perspectives:

1) An external perspective (or) *context model*- we model the context (or) environment of the system.
2) An interaction perspective (or*) interaction model*- when you model the interaction between a system and its environment (or) between the components of a system.
3) A structural perspective (or*) structural model*- we model the organization of system (or) the structure of the data that is processed by the system.
4) A behavioral perspective (or*) behavioral model*- when you model the dynamic behavior of the system and how it responds to events.

We use the following UML diagrams to create different kind of models:

1) Activity diagrams- show the activities involved in a process (or) in data processing.
2) Use case diagrams – show the interactions between a system and its environment.
3) Sequence diagrams- show interactions between actors and the system and between system components.
4) Class diagrams- show the classes in the system and its associations between these events.
5) State diagrams- show how the system reacts to internal and external events.

**Context models:** the context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. At the early stage in the specification of a system, we should decide on the system boundaries. This involves working with stakeholders to decide what functionality should be included in the system and what is provided by the system's environment. So that decisions should be made early inthe process to limit the system cost and time needed for understanding the system requirements and design. In MHC-PMS, the advantage of relying on other systems for patient information is that we avoid duplicating data. The disadvantage using other systems may make it slower to access information.

**Interaction models:**

There are two approaches to interaction modeling: use case diagrams, sequence diagrams.

All systems involve interaction of some kind. This can be user interaction, interaction between the system being developed and other systems, and between components of the system. Modeling user interaction heighlights the communication problem that may arise. Modeling component interation helps us understand if a proposed system structure is likely to deliver the required performance and dependability.

Use case modeling was developed by Jacobson. This is widely used to support requirements elicitation. **Use case diagrms** give faily simple overview of an interaction. A use case can be taken as a simple scenario that describes what user expects from a system. Each use case represents a discrete task that involves external interaction with a system. A use cse is shown as an ellipse with actors as stick figures. Stick figure indicates human/ external system/ hardware interaction. **Sequence diagrams** in the UML are primarily used to model the interaction between the actors and the objects in a system and the interactions between the objects themselves. A sequence diagrms shows the sequence of interactions that take place during a particular use case (or) use case instance. In our example diagram models the interactions involved in the View patient information use case, where medical receptionist ca see patient information. Objects/actors are listed along the top of the diagram, with dotted line drawn vertically from these. Interations between objects are indicated by annotated arrows. The rectangle on the dotted lines indicates the lifeline of the object concerned. You read the sequence of interaction from top to bottom. The annotations on the arrows indicate the calls to the object, their parameters, and the return values.

**Structural Models: t**hese display the organization of a system in terms of the components that make up the system and their relationships. Here, **class diagrams** are used when modeling object oriented system. Class diagram gives structural model to show the classes in a system and associations between the classes. When you are developing a model, the first stage is usually to look at the world, identify objects, and represent these as classes. The simplest way of writing these is to write the class name in a box. You can also note the existence of an association by drawing a line between classes.

**Behavioral Models:** these are models of the dynamic behavior of the system as it is executing. They show what happens (or) what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1) Data- some data arrives that has to be processed by the system. ( Data driven modeling)- here, data flow diagrms are used. ( UML does not support these diagrams).
2) Events- some event happens that triggers system processing. Events may have associated data but this is not always the case.

UML supports event-based modeling using **state diagrams**. State diagrams show a finite system states and events that cause transitions from one state to another. Event modeling shows how a system responds to external/internal events.

## System Design ( UNIT-III)

### 7. Design Concepts:

***Abstraction***- at the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

A procedural abstraction refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word “open” for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A data abstraction is a named collection of data that describes a data object. The data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

***Architecture***- 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. This rendering serves as a framework from which more detailed design activities are conducted.

***Patterns***- a design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces”. The intent of each design pattern is to provide a description to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

**Separation of Concerns-** it suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

**Modularity-** is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

Rather than monolithic software (i.e., a large program composed of a single module), you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

**Information Hiding-** modules should be specified and designed so that information (algorithms and data) 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.

**Functional Independence-** the concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules of software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

**Refinement-** a program is developed by successively refining levels of procedural detail. Refinement is actually a process of elaboration. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

**Aspects-** as requirements analysis occurs, a set of “concerns” is uncovered. Some of these concerns span the entire system and cannot be easily compartmentalized. Consider two requirements, A and B. Requirement A crosscuts requirement B “if a software decomposition has been chosen in which B cannot be satisfied without taking A into account. An aspect is a representation of a crosscutting concern. In an ideal context, an aspect is implemented as a

separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

**Refactoring-** refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. 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.

**Short Notes:**

A **cohesive module** performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. **Coupling** is an indication of interconnection among modules in a software structure. In software design, you should strive for the 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 propagate throughout a system.

**Different ways of writing a system requirements:** natural languge sentences-requirments are written using numbered sentences in natural language, structured natural language- uses a programming language to write requirements, graphical notations- graphical models are used to define functional requirements for the system, mathematical specifications- mathematical concepts such as finite-state machines, sets are used to write requirements with no ambiguity.

**Structure of the Software requirements document (or) SRS:** it consists of preface, introduction, glossary, user requirements definition, system architecture, system requirements specification, system models, system evolution, appendices, index.

**Activity diagrams:** are intended to show the activites that make up a system process and the flow of control from one activity to another. Start is indicated by filled circle; end is indicated by filled circle inside another circle. Rectangles with round corners represent activities. Arrows represent the flow of work from one activity to another. A solid bar is used to indicate activity co-ordination. Arrows may be annotated with guards.

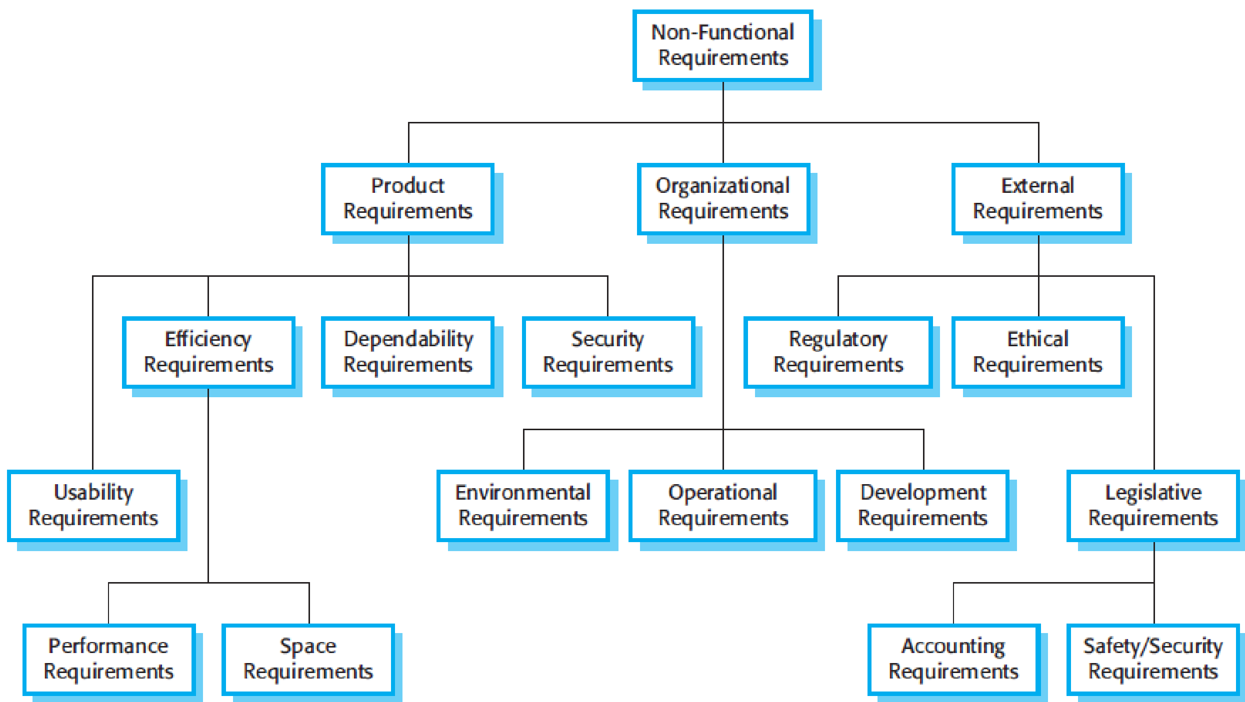



Fig: types of non-functional requirements

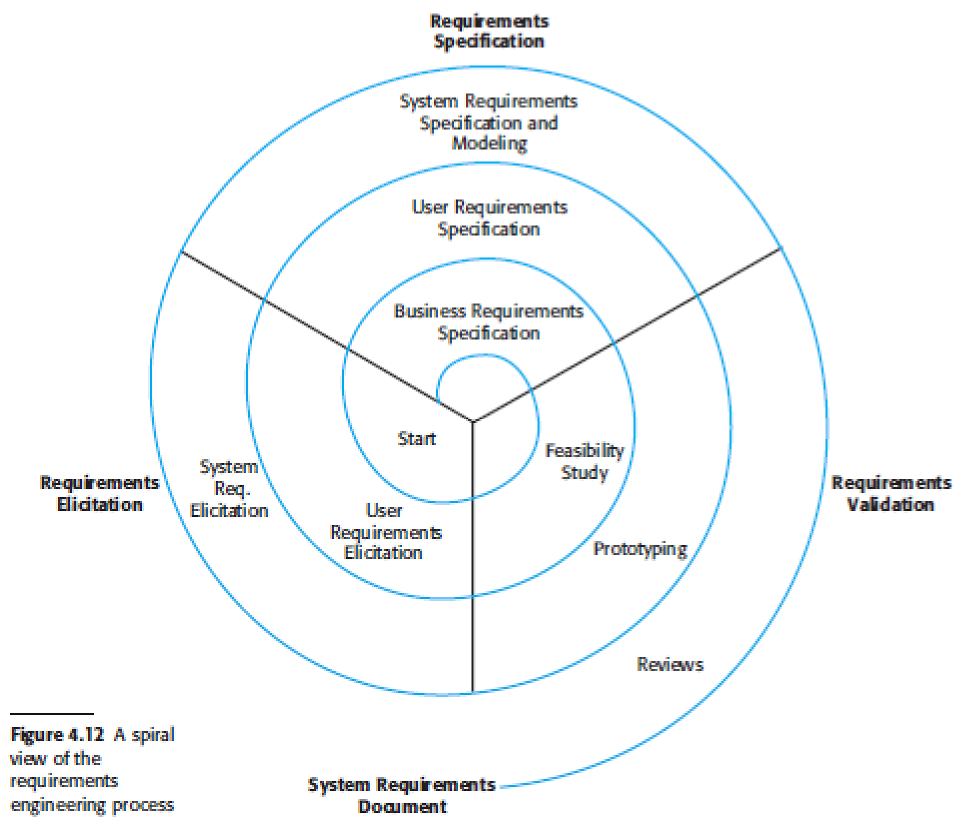



**Figure 4.12** A spiral view of the requirements engineering process

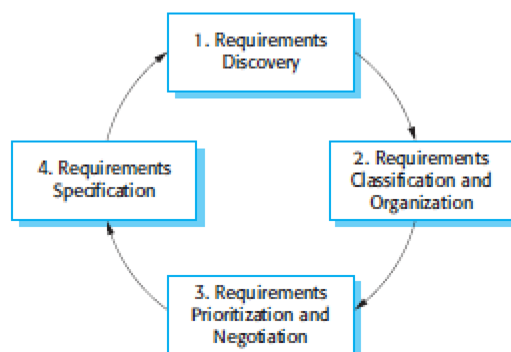



**Figure 4.13** The requirements elicitation and analysis process

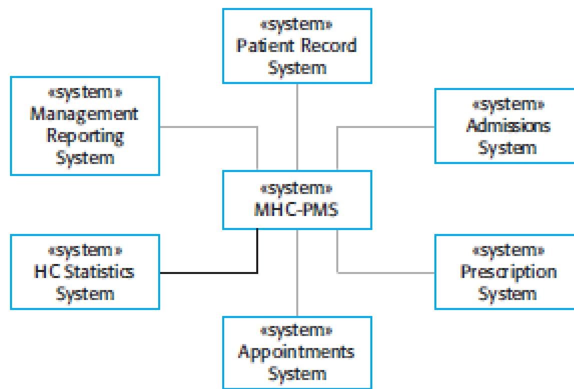



Fig: context model of MHC-PMS

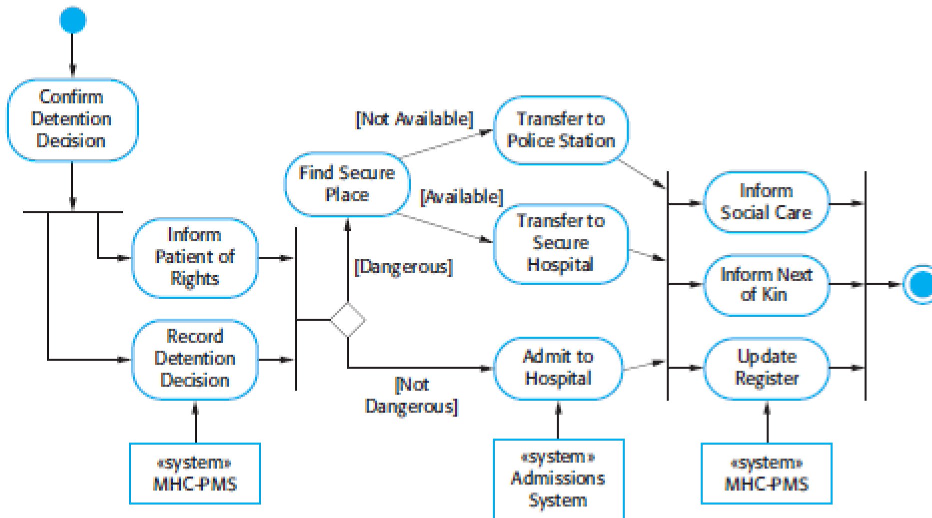



Fig: activity diagram of detention of mental patient

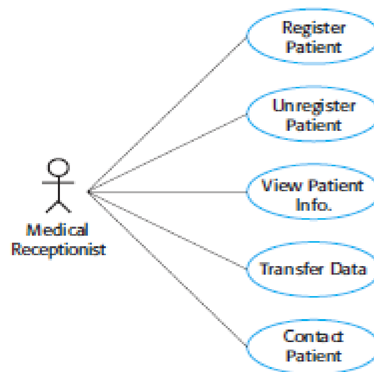



Fig: Use case diagram- use cases of Medical Receptionist

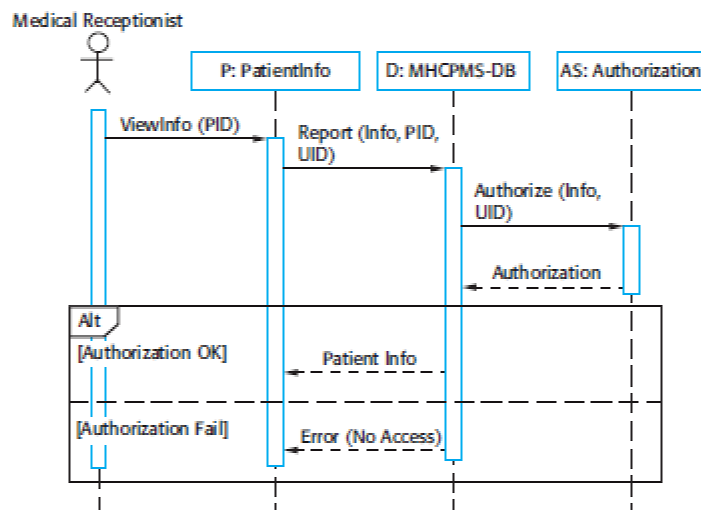



**Figure 5.6** Sequence diagram for View patient information

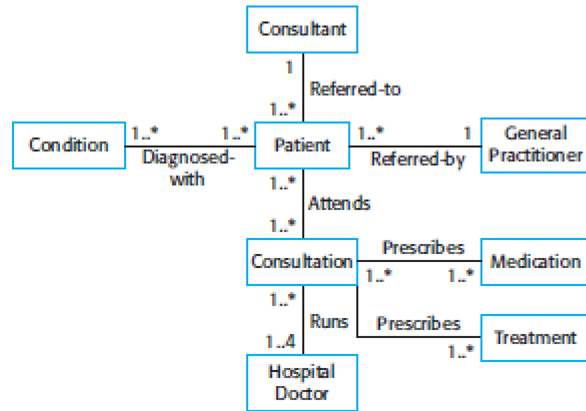



Fig: class diagram of MHC-PMS

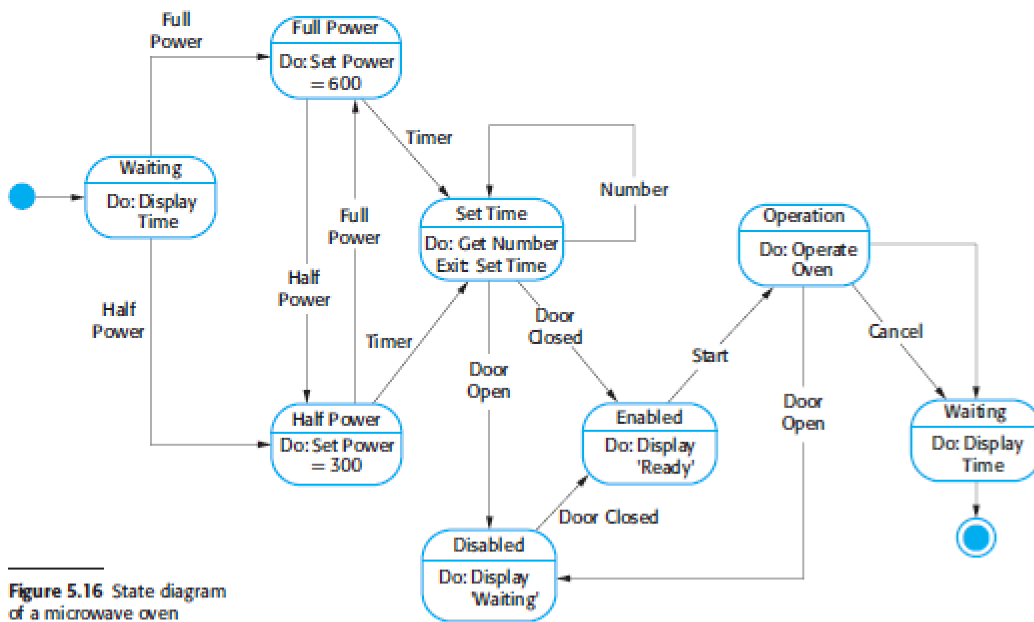



**Figure 5.16** State diagram of a microwave oven

# (UNIT-III)

**( Topics on Architectural design, Component-level design needs to given later)**

## 1) The Golden rules for user interface design ( UI Design)

i. Place the user in control.

ii. Reduce the user's memory load.

iii. Make the interface consistent.

**i). Place the user in control.**

**a) Define interaction modes in a way that does not force a user into unnecessary actions.** For example, if spell check is selected in a word-processor menu, the user should be able to enter and exit the mode with little or no effort.

**Provide for flexible interaction.** Because different users have different interaction preferences, for example, software might allow a user to interact via keyboard commands, mouse movement.

**Allow user interaction to be interruptible.** the user should be able to interrupt the sequence to do something else. The user should also be able to "undo" any action.

**Hide technical internals from the casual user.** The user should not be aware of the operating system, file management functions.

**ii). Reduce the user's memory load.**

The more a user has to remember, the more error-prone the interaction with the system will be. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

**Establish meaningful defaults.** The initial set of defaults should make sense for the average user. However, a "reset" option should be available, enabling the redefinition of original default values.

**Define shortcuts that are intuitive.** When mnemonics are used (e.g., alt-P to do the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

**iii). Make the Interface Consistent**

The interface should present and acquire information in a consistent fashion.

**Allow the user to put the current task into a meaningful context.** It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. The user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

**Maintain consistency across a family of applications.** A set of applications should all implement the same design rules so that consistency is maintained for all interaction.

**2) Interface Analysis and Design Models**

The overall process for analyzing and designing a user interface begins with the creation of different models of system function. Four different models come into play when a user interface is to be analyzed and designed.

The software engineer establishes a user model and it establishes the profile of end users of the system. The software engineer creates a design model, the end user develops a mental image that is often called the user's mental model or the system perception, and the implementers of the system create an implementation model. The user's mental model is the image of the system that end users carry in their heads. The implementation model combines the outward manifestation of the computerbased system (the look and feel of the interface). Each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

**User interface design process:**

The analysis and design process for user interfaces is iterative and can be represented using a spiral model. The user interface analysis and design process encompasses four distinct framework activities: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping-the only practical way to validate what has been designed.

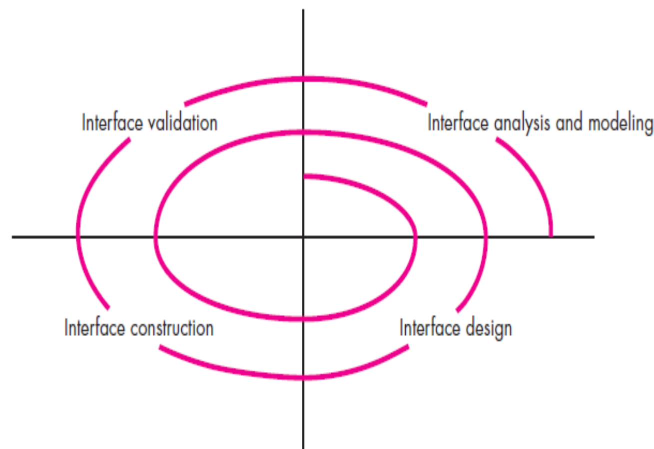
Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral). Finally, analysis of the user environment focuses on the physical work environment.

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.




**Interface analysis:**

User interface design, understanding the problem means understanding (1) the end users who will interact with the system, (2) the tasks that end users must perform, (3) the content that is presented, and (4) the environment in which these tasks will be conducted. **User analysis**- the only way that you can get the user mental image and the design model to converge is to understand the users themselves as well as how these people will use the system. Information from sources such as User interviews, sales input, marketing input, support input; is used for user analysis. **Task Analysis and Modeling-** the use case is developed to show how an end user performs some specific work-related task. This use case provides a basic description of one important work task for the system. From it, you can extract tasks, objects, and the overall flow of the interaction. **Analysis of Display Content-** the output data objects that are produced by an application. These data objects may be (1) generated by components, (2) acquired from data stored in a database that is accessible from the application. **Analysis of the Work Environment-** In some applications the user interface is placed in a "user-friendly location" (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

**3) Interface design steps**

Once interface analysis has been completed, all objects and actions required by the end user have been identified in detail and the interface design activity commences. Interface design is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.

The following are the steps for designing user interface model:

1. Using information developed during interface analysis, define interface objects and actions.
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

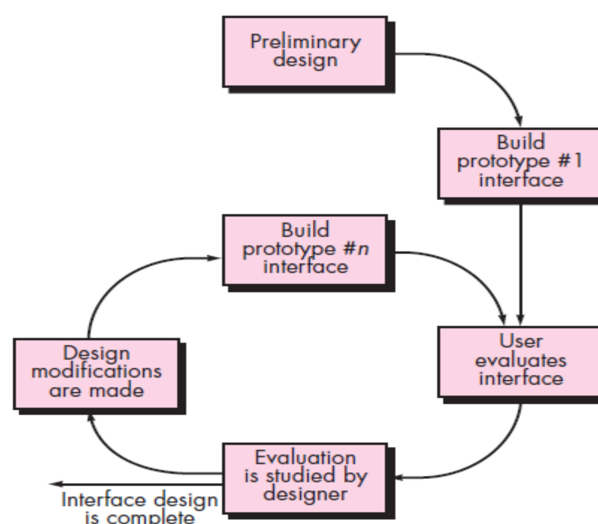
You should (1) always follow the golden rules , (2) model how the interface will be implemented, and (3) consider the environment (e.g., display technology, operating system, development tools) that will be used.

**Applying Interface Design Steps**

The definition of interface objects and the actions is an important step in interface design. To do this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon). The implication of this action is to create a hard-copy report. When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.

**4) User interface design evaluation:**




Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. The user interface evaluation cycle takes the form shown in above Figure. After the design model has been completed, a first-level prototype is created. The prototype

is evaluated by the user, who provides you with direct comments about the interface.

You can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be: (1) simple yes/no response, (2) numeric response, (3) percentage (subjective) response. If quantitative data are desired, a form of time-study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent- are collected and used as a guide for interface modification.

Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. If you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten.

**5) Design Issues**

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. It is far better to establish each at the beginning of software design, when changes are easy and costs are low.

**Response time.** In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action. System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds.

**Help facilities.** Almost every user of an interactive, computer-based system requires help now and then. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

**Error handling.** Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

**Menu and command labeling.** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type alt-D when a graphics object is to be duplicated in one

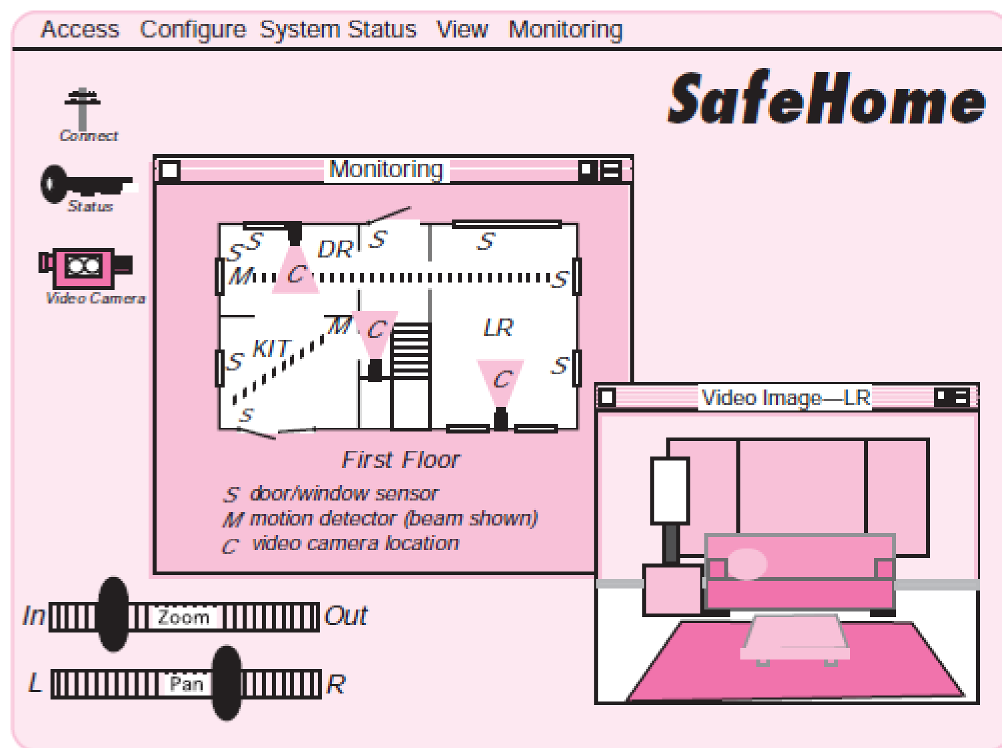
application and alt-D when a graphics object is to be deleted in another. The potential for error is obvious.

**Application accessibility.** As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users who may be physically challenged.

**Internationalization.** The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. Localization features enable the interface to be customized for a specific market.

**6). An Example System-SafeHome system:**

It is Preliminary screen layout-




**Preliminary use case:** I want to gain access to my SafeHome system from any remote location via the Internet. Using browser software operating on my notebook computer (while I'm at work or traveling), I can determine the status of the alarm system, arm or disarm the system, reconfigure security zones, and view different rooms within the house via preinstalled video cameras.

To access SafeHome from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change the status by arming or disarming SafeHome. I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

To provide a brief illustration of the design steps noted previously, consider a user scenario for the

SafeHome system. The following homeowner tasks, objects, and data items are identified:
• *accesses* the SafeHome system, • *enters* an **ID** and **password** to allow remote access
• *checks* **system status**, • *arms* or *disarms* SafeHome system
• *displays* **floor plan** and **sensor locations**, • *displays* **zones** on floor plan
• *changes* **zones** on floor plan, • *displays* **video camera locations** on floor plan
• *selects* **video camera** for viewing, • *views* **video images** (four frames per second)
• *pans* or zooms the **video camera**

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).

A preliminary sketch of the screen layout for video monitoring is created. To invoke the video image, a video camera location icon, C, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

**Note:**

You are suggested to Prepare an example Swimlane diagram.

## UNIT-III

**( Design Conepts, Architectural design, Component-level design) (Note: Diagrams given in the ending pages. )**

### 1) Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. **Analysis Classes- t**he requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem. The level of abstraction of an analysis class is relatively high.

**As design model evolves**, we will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software.

**Five different types of design classes**, each representing a different layer of the design architecture, can be developed: • User interface classes define all abstractions that are necessary for human computer interaction (HCI).

• Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain. • Process classes implement lower-level business abstractions required to fully manage the business domain classes.

• Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software. • System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. **Four characteristics** of a well-formed design class:

**Complete and sufficient.** A design class should be the complete encapsulation of all attributes and methods. Sufficiency ensures that the design class contains only those methods that are sufficient, no more and no less.

**Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

**High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.

**Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. If a design model is highly coupled the system is difficult to implement, to test, and to maintain over time.

**2) The Design Model**

The design model can be viewed in **two different dimensions** shown in **Figure 1**. The process dimension indicates the evolution of the design model as design tasks are executed. 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. In Figure 1 , the dashed line indicates the boundary between the analysis and design models.

The elements of the design model use the UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

In Figure, the model elements indicated 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 is usually delayed until the design has been fully developed.

**You can apply design patterns** at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

**Data-design elements-** data design (or data architecting) creates a model of data that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations.

**Architectural Design Elements-** The architectural model is derived from three sources: information about the application domain; (2) specific requirements model elements such as data flow diagrams, analysis classes, their relationships and collaborations; and (3) the availability of architectural styles and patterns. The architectural design element is usually depicted as a set of interconnected subsystems. Each subsystem may have it's own architecture.

**Interface design elements:** The interface design for software is similar to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components. There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, and (3) internal interfaces between various design components.

**Component-Level Design Elements-** The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. The component-level design describes the internal detail of each software component. The component-level design defines data structures, algorithmic detail for all processing, and an interface. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be

represented using either pseudocode or flowchart.

**Deployment-Level Design Elements-** indicate how software functionality and subsystems will be allocated within the physical computing environment. During design, a UML deployment diagram is developed and then refined. It shows the computing environment but does not explicitly indicate configuration details.

## Architectural Design

### 3) Taxonomy of Architectural Styles

Architectures are shown in figure 3(a), (b), (c), (d).

**Each style** describes a system category that encompasses (1) a set of components (e.g., a database, computational modules); (2) a set of connectors that enable "communication, coordination and cooperation" among components; (3) constraints that define how components can be integrated; (4) semantic models that enable a designer to understand the overall properties of a system.

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

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

**Call and return architectures:** This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. Main program nvokes a number of program components that in turn may invoke still other components.

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

**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 are accomplished via message passing.

**4) Architectural Genres** genre implies a specific category within the overall software domain. Grady Booch suggests the following architectural genres for software-based systems:

Artificial intelligence—Systems that simulate human cognition, locomotion; Commercial and nonprofit—Systems that are fundamental to the operation of a busines; Communications—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data; Content authoring—Systems that are used to create or manipulate textual or multimedia artifacts; Devices—Systems that interact with the physical world to provide some point service for an individual; Entertainment and sports—Systems that manage public events or that provide a large group entertainment experience; Financial—Systems that provide the infrastructure for transferring and managing money; Games—Systems that provide an entertainment experience for individuals or groups; Government—Systems that support the conduct and operations of a local, state, federal, global, or other political entity. **Other genres are** industrial, legal, medical, militaty, operating

**Each genre** represents a unique challenge. As an example, consider the software architecture for a game system. Game systems require the computation of intensive algorithms, sophisticated computer graphics, streaming multimedia data sources, real-time interactivity via conventional and unconventional inputs.

**Architectural patterns** address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design. A specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call-and-return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns.

**5) Architectural Design Steps**

Examples are given in Figures 4, 5, 6, 7.

**a) Representing the System in Context**- 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 consists of systems that interoperate with the "target system"are represented as: 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—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles). In Figure, SafeHome security function ACD is given. Example ACD is given in **Figure 2**.

**b) Defining Archetypes**

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an

architecture for the target system. The target system architecture is composed of these archetypes, which represent stable elements of the architecture. Archetypes can be derived by examining the analysis classes. For our example the SafeHome home security function, you might define the following archetypes:

- Node. Represents 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 (output) indicators.
- Detector. An abstraction that encompasses all sensing equipment that feeds information into the target system.
- Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- Controller. An abstraction that depicts the mechanism that allows the arming or disarming of a node. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. The above archetypes are illustrated in Figure.

**c) Refining the Architecture into Components**

As the software architecture is refined into components, the structure of the system begins to emerge. Each of these top-level components would have to be elaborated iteratively. The design details of all attributes and operations would not be specified until component-level design. The overall structure is shown in UML component diagram in Figure. Transactions are acquired by external communication management as they move in from components that process the SafeHome GUI and the Internet interface. This information is managed by a SafeHome executive component that selects the appropriate product function (in this case security). The control panel processing component interacts with the homeowner to arm/disarm the security function. The detector management component polls sensors to detect an alarm condition, and the alarm processing component produces output when an alarm is detected.

**d) Describing Instantiations of the System**

The major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary. To accomplish this, an actual instantiation of the architecture is developed. By this We mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

## Component-Level Design

**Component-level design** occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. It is possible to represent the component-level design using a programming language. An alternative approach is to represent the component-

level design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code.

A **component** is a modular building block for computer software. The OMG Unified Modeling Group defines a component as "... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

**6) An Object-Oriented View**

In object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations. all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, you begin with the requirements model and elaborate analysis classes (relate to the problem domain) and infrastructure classes (relate to support services for the problem domain). The data structures, interfaces, and algorithms defined should conform to a variety of well-established design guidelines.

To illustrate this, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called PrintJob was derived. During architectural design, PrintJob is defined as a component. In the Component-level design, the details of the component PrintJob must be elaborated to provide sufficient information to guide implementation. The interfaces computeJob and initiateJob imply communication and collaboration with other components. For example, the operation computePageCost() might collaborate with a PricingTable component that contains job pricing information. Finally, the mechanisms required to implement the interface are designed. For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system. Example elaboration of a design component is shown in **Figure 9**.

**Traditional View:**

During component-level design, each module is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed. The behavior of the module is sometimes represented using a state diagram. To illustrate this, consider the module ComputePageCost. The intent of this module is to compute the printing cost per page based on specifications provided by the customer. Data required to perform this function are: number of pages in the document, total number of documents to be produced, one- or two-side printing, color requirements, and size requirements. These data are passed to ComputePageCost via the module's interface. ComputePageCost uses

these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job. Component-level design for ComputePageCost is shown in **Figure 8**.

**7) Basic Design Principles.**

**The Open-Closed Principle (OCP). "**A moule [component] should be open for extension but closed for modification"**.** You should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself.

**The Liskov Substitution Principle (LSP).** "Subclasses should be substitutable for their base classes". It suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.

**Dependency Inversion Principle (DIP).** "Depend on abstractions. Do not depend on concretions". Abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components, the more difficult it will be to extend.

**The Interface Segregation Principle (ISP).** " Many client-specific interfaces are better than one general purpose interface". ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client.

**The Common Closure Principle (CCP).** "Classes that change together belong together." Classes should be packaged cohesively. When classes are packaged as part of a design, they should address the same functional or behavioral area.

**The Common Reuse Principle (CRP).** "Classes that aren't reused together should not be grouped together". When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package must now update to the most recent release of the package and be tested.

**8) Conducting Component-Level Design**

Component-level design is elaborative in nature. You must transform information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. Examples are given in Figures 10, 11 and 12.

**The following steps are for component-level design**, when it is applied for an object-oriented system.

**Step 1. Identify all design classes that correspond to the problem domain.** Using the requirements and architectural model, each analysis class and architectural component is elaborated.

**Step 2. Identify all design classes that correspond to the infrastructure domain.** Classes and

components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

**Step 3. Elaborate all design classes that are not acquired as reusable components.** Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered. conducted.

**Step 3a. Specify message details when classes or components collaborate.**

The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. It is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system.

**Step 3b. Identify appropriate interfaces for each component.** An interface is the equivalent of an abstract class that provides a controlled connection between design classes. Every operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

**Step 3c. Elaborate attributes and define data types and data structures required to implement them.** In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation.

**Step 3d. Describe processing flow within each operation in detail.** This may be accomplished using a programming language-based pseudocode or with a UML activity diagram.

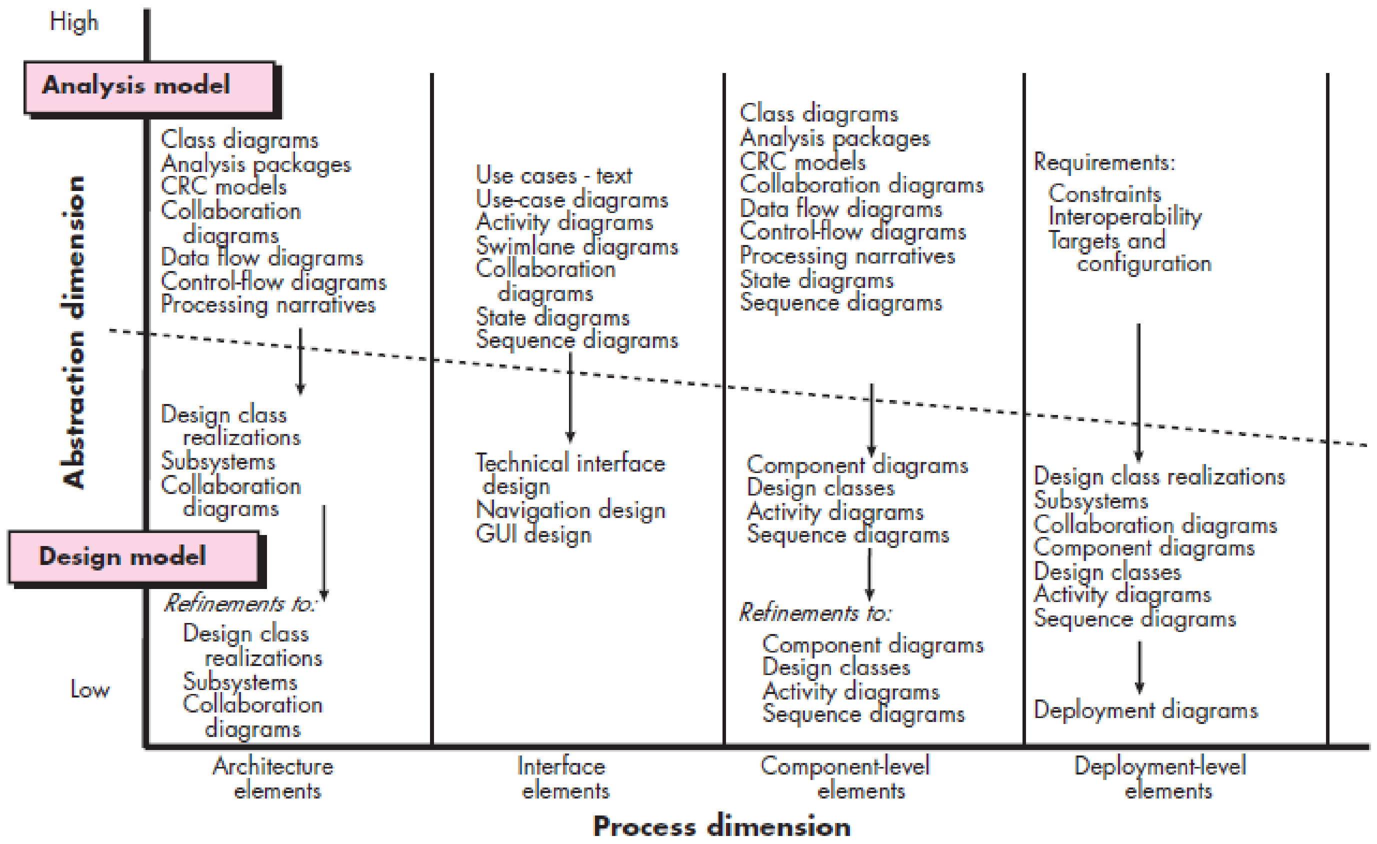
**Step 4. Describe persistent data sources and identify the classes required to manage them.** In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

**Step 5. Develop and elaborate behavioral representations for a class or component.** UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system. During component-level design, it is sometimes necessary to model the behavior of a design class.

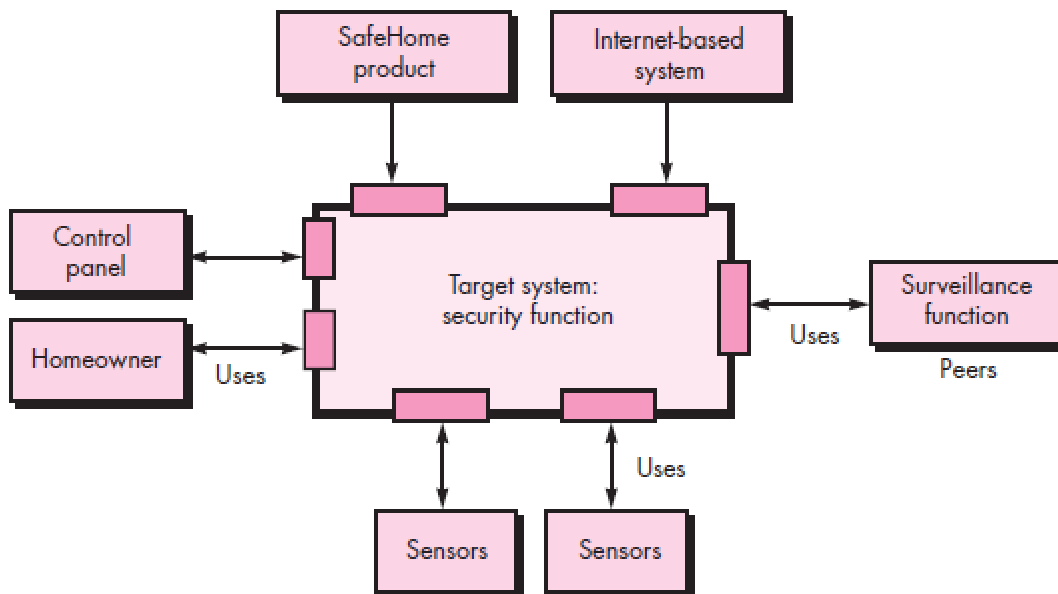
**Step 6. Elaborate deployment diagrams to provide additional implementation detail.** During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. In some cases, specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

**Step 7. Refactor every component-level design representation and always consider alternatives.** The first component-level model you create will not be as complete, consistent, or accurate as the nth iteration you apply to the model. It is essential to refactor as design work is conducted.

**1) Design process model:**

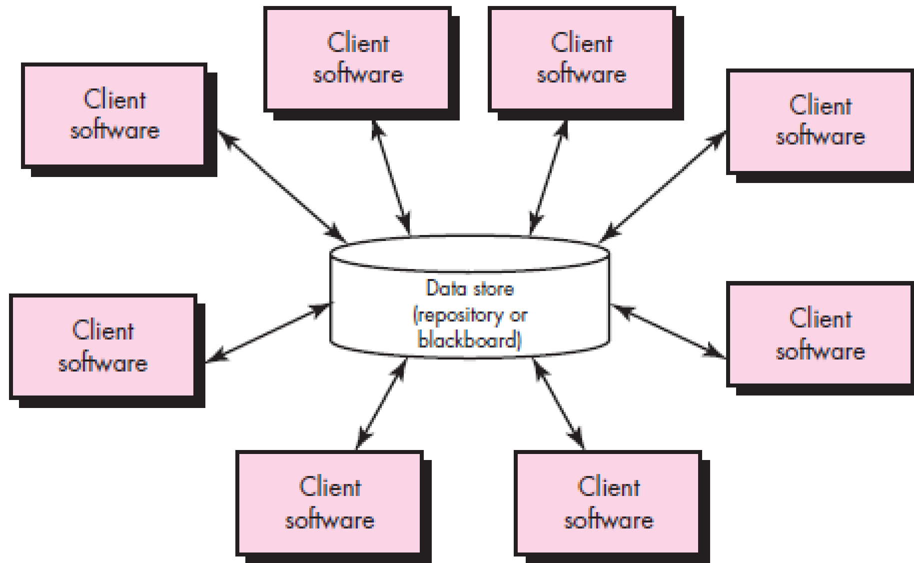



**2) Architectural context diagram for SafeHome software system:**

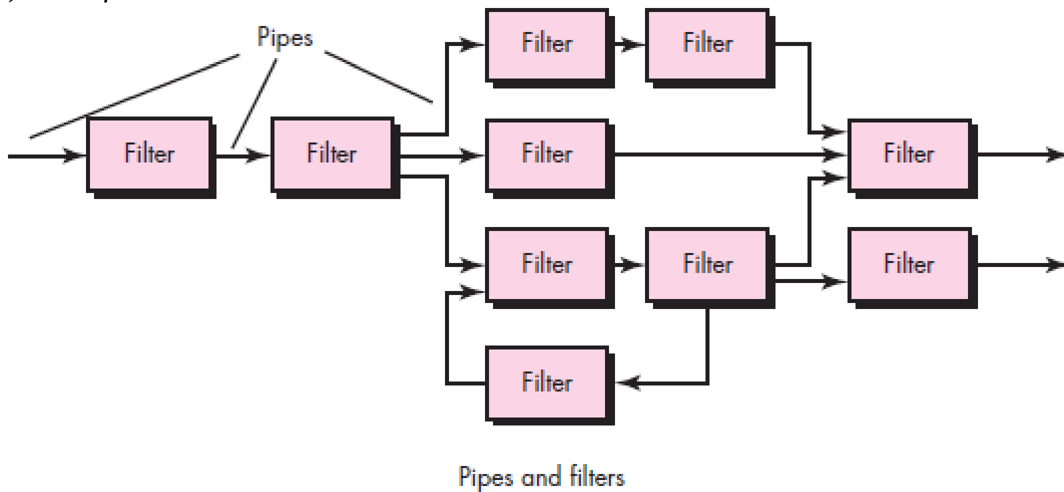


**3) Taxonomy / Classification of Architectural Styles:**

*a) Data- centered architecture*

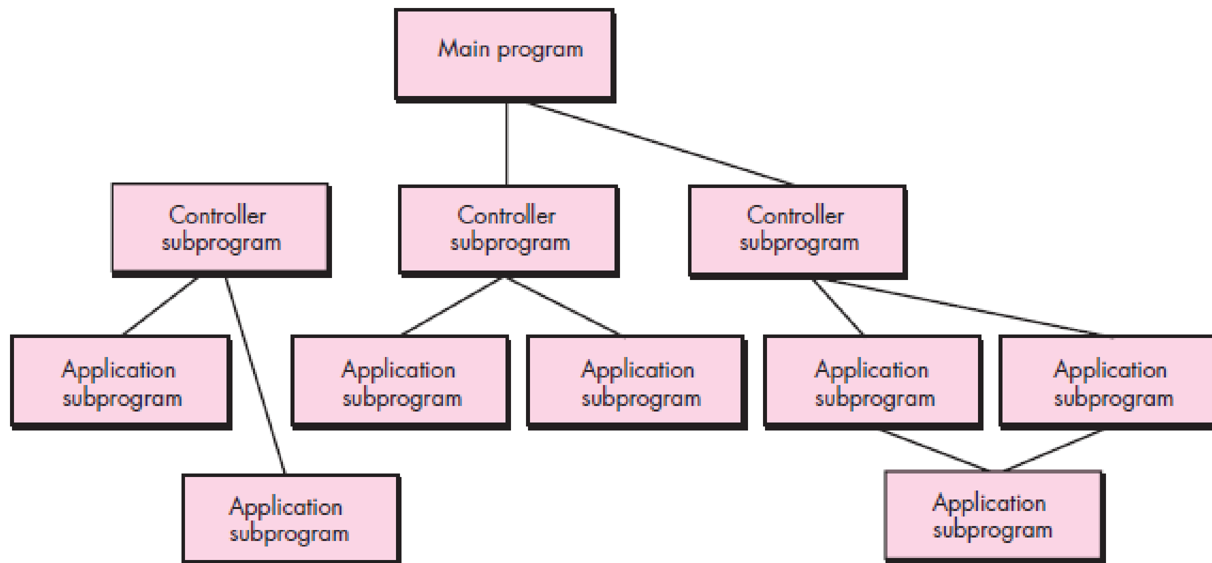



*b) Data-flow centered architecture*

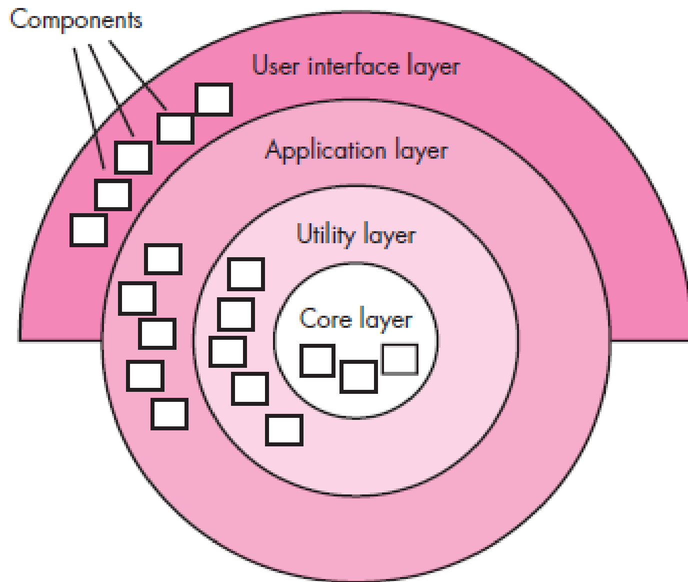



Pipes and filters

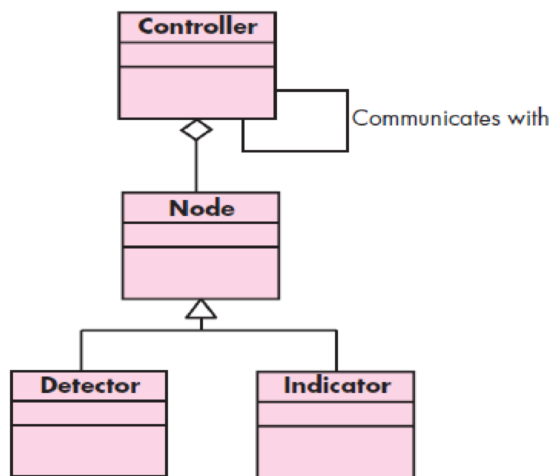
*c) Call and return architeture*



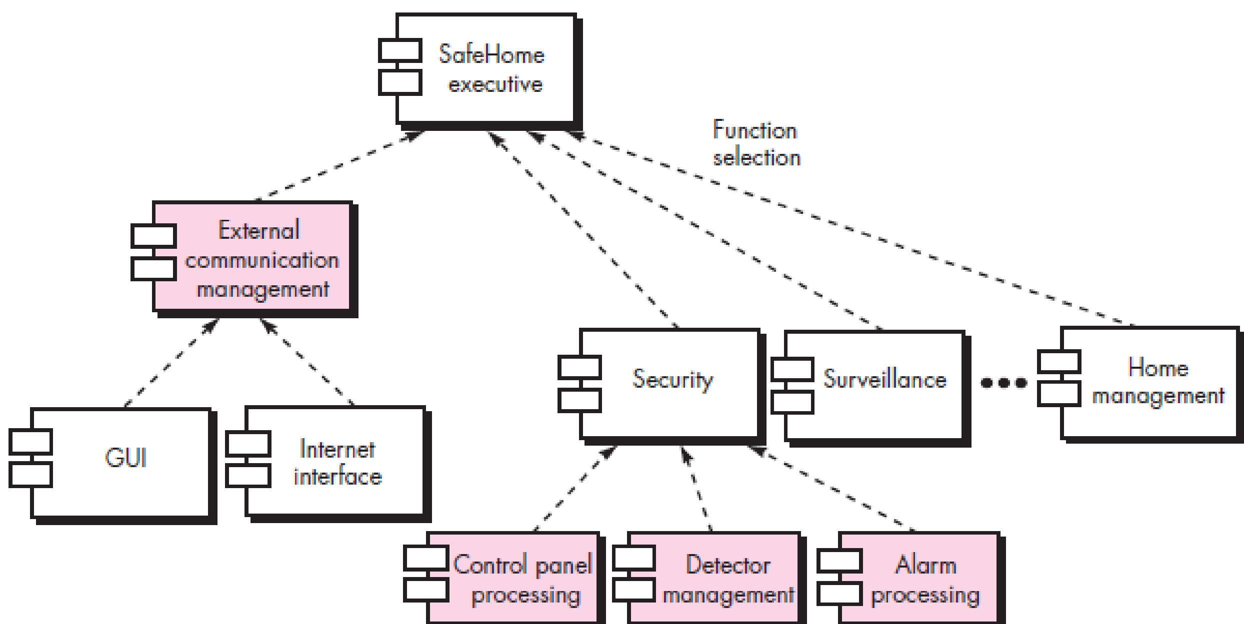
*d) Layered architecture*



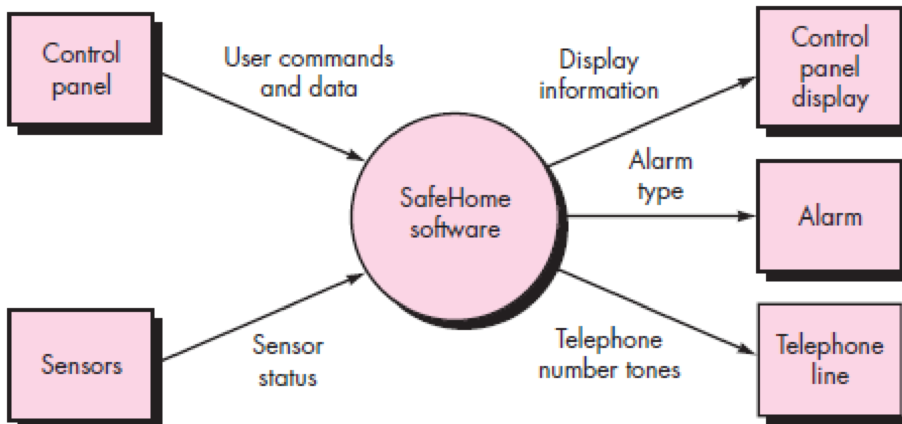

**4)** UML relationships for SafeHome security function archetypes



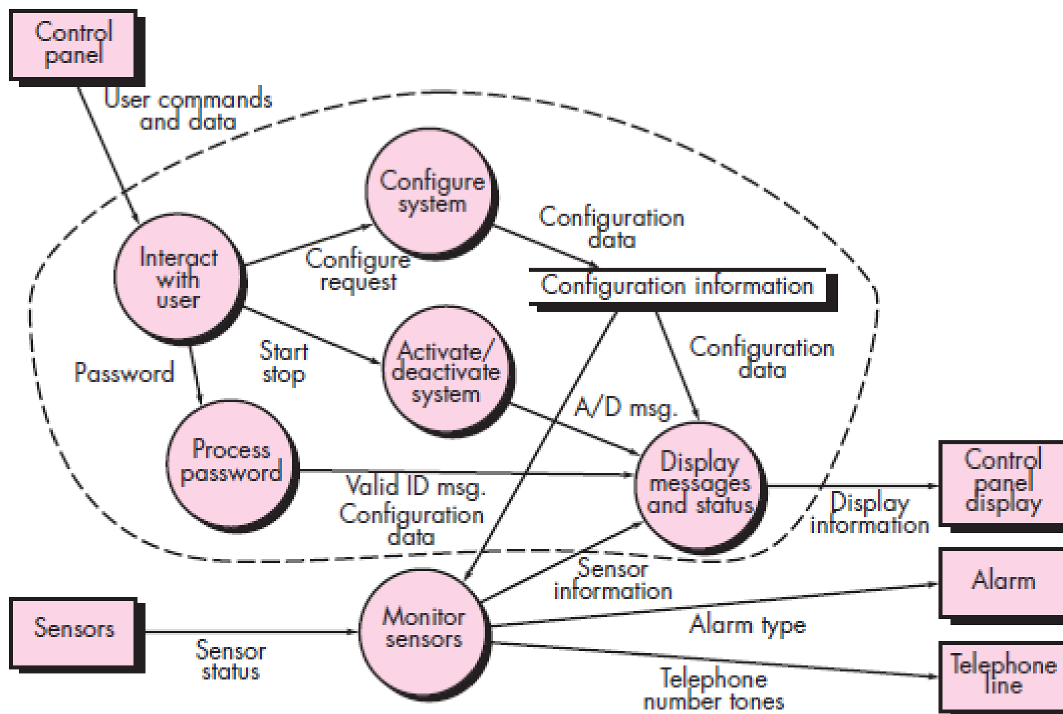

**5)** Overall architectural structure for SafeHome with top-level components



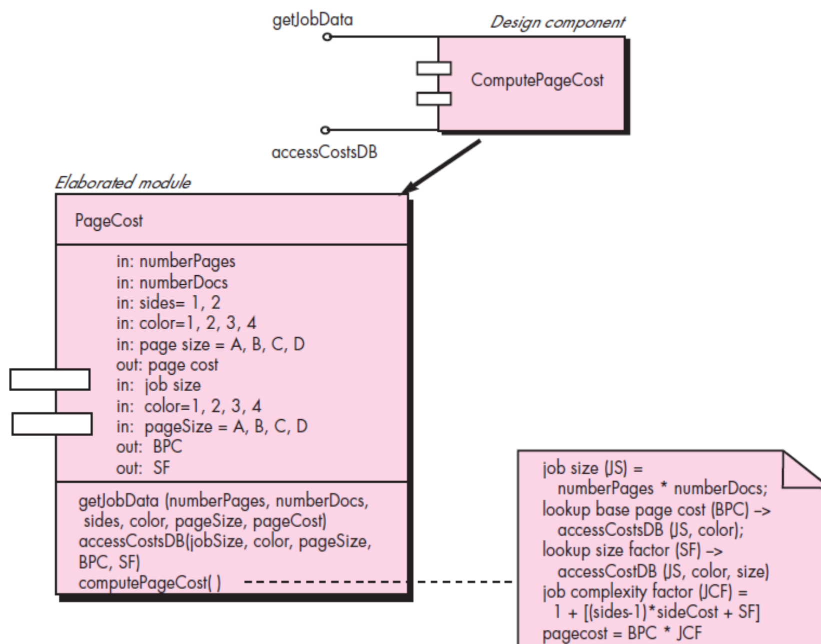
**6)** Context-level DFD for the SafeHome security function



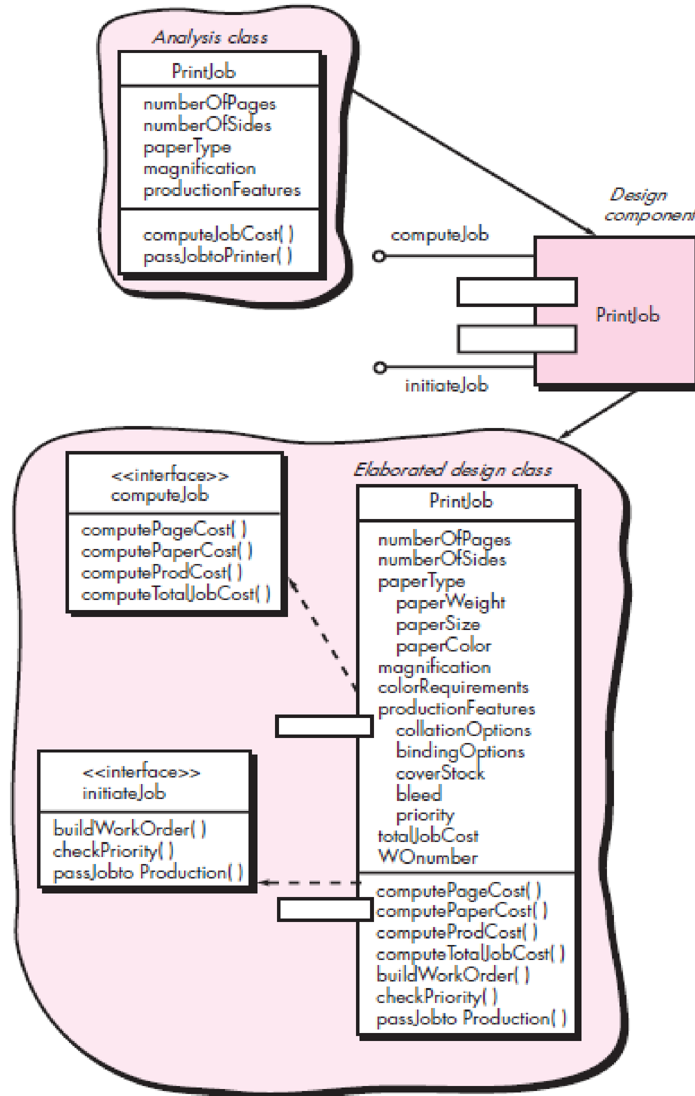

**7)** Level 1 DFD for the SafeHome security function



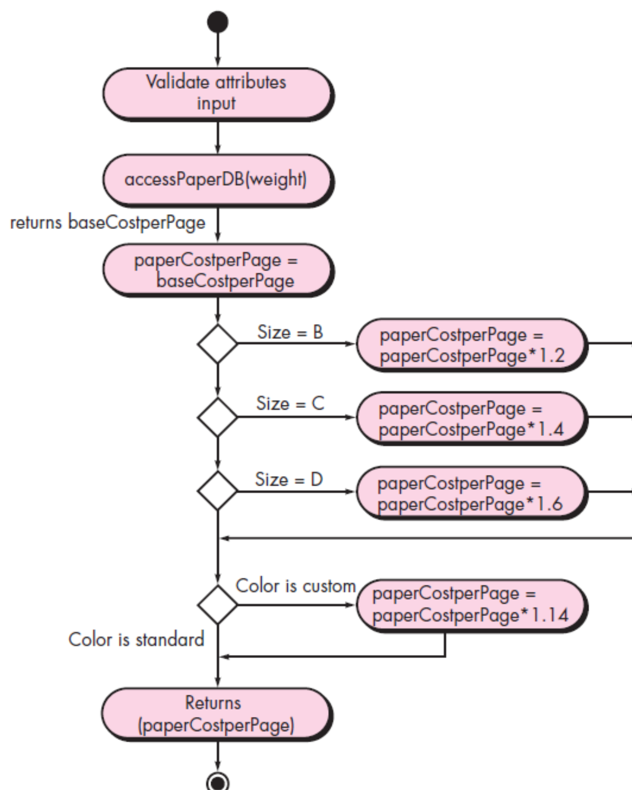

**8)** Component-level design for ComputePageCost



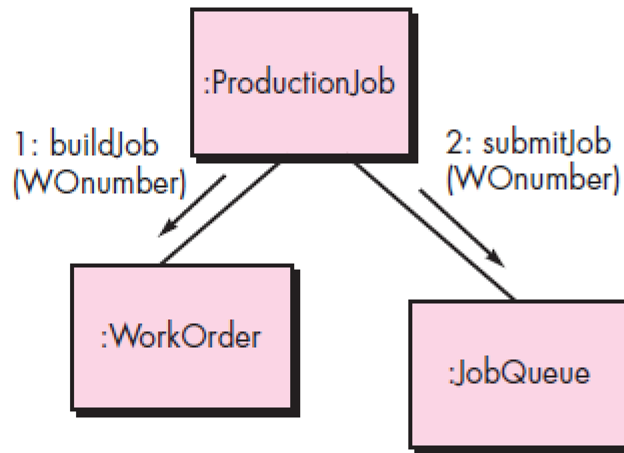
**9) Component-level Desgin:** Elaboration of a design component



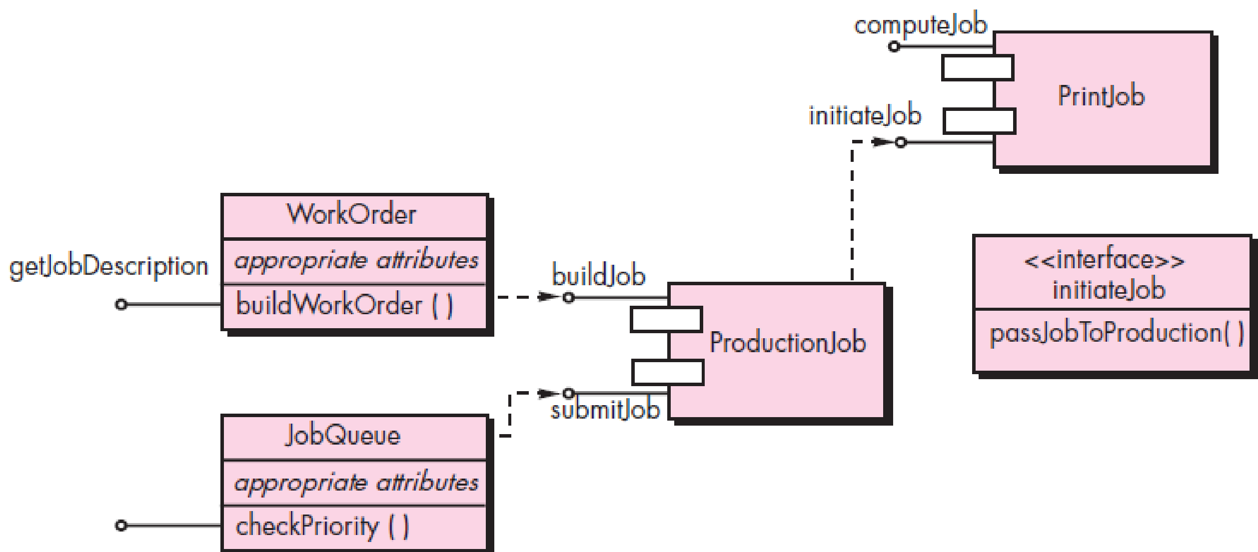

**10)** UML activity diagram for compute-PaperCost()



**11)**Collaboration diagram with messaging




**12)** Refactoring interfaces and class definitions for PrintJob



# Software testing ( Unit-IV)

**( Note: Discussion on various kinds of metrics will be given along with Unit-V notes)**

**Software testing strategy** provides a road map that describes the steps to be conducted and how much effort, time, and resources will be required for testing. Any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

**Generic characteristics** of software testing strategies:

1) To perform effective testing, you should conduct effective technical reviews

2) Testing begins at the component level and works “outward” toward the integration of the system.

3) Testing is conducted by the developer of the software and (for large projects) an independent test group.

4) Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

**1) Verification and Validation:**

Software testing is often referred to as verification and validation ( V&V). Verification means the set of tasks to ensure that software correctly implements a specific function. Validation means a different set of tasks to ensure that the software is traceable to customer requirements.

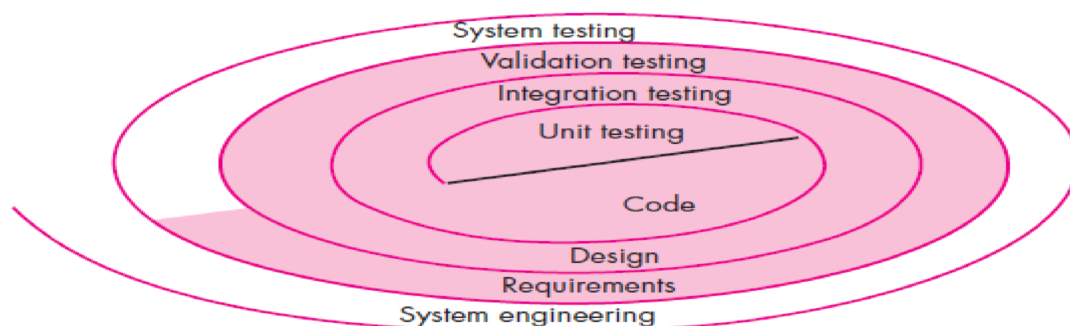
Boehm [Boe81] states : Verification: “Are we building the product right?” , Validation: “Are we building the right product?”

Verification and validation includes a wide array of Software Quality Assurance activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing.

**2) Software Testing Strategy :**

The software process may be viewed as the spiral in Figure given below. To develop computer software, you spiral inward. Initially, system engineering defines the role of software and leads to software requirements analysis. Moving inward along the spiral, you come to design and finally to coding.

To test computer software, you spiral out in a clockwise direction along streamlines. Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software. Unit test focuses on each component individually and ensures that it



functions properly as a unit. Integration testing, where the focus is on design and the construction of the software architecture. Validation testing, where requirements are validated against the software. Validation testing provides assurance that software meets all informational, functional, behavioral, and performance requirements. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements are properly connected and that overall system function/performance is achieved. It consists of the software and other system elements are tested as a whole.

**3) Validation testing:**

Whether it is conventional software or object-oriented software or Web apps, no diference in the testing strategy. If a software requirements specification has been developed, it contains validation criteria that forms the basis for a validation-testing approach.

Software validation is achieved through a series of tests. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability). After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic is accepted or (2) a deviation from specification is found and a deficiency list is created.

An important element of the validation process is a configuration review (audit). The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged.

**Alpha and Beta Testing**

Software develpers use a process called alpha and beta testing to uncover errors that the end user able to find. The **alpha test** is conducted at the developer’s site by a group of representative users. The software is used in a natural setting, recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The **beta test** is conducted at one or more end-user sites. While doing alpha testing, the developer generally will not present. It is a “live” application of the software in an environment. The customer records all problems (real or imagined) that are encountered during beta testing and reports to the developer.

A variation on beta testing, called **customer acceptance testing**, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software.

**4) System testing:**

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

**Recovery Testing** computer-based systems must recover from faults and resume processing with little or no downtime. A system must be fault tolerant- faults must not cause overall system function to cease. Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits. **Security Testing** any computer-based system that manages sensitive information is a target for improper or illegal penetration. Security testing attempts to verify that protection mechanisms built into a system will protect it from improper penetration.

**Stress Testing** Stress tests are designed to confront programs with abnormal situations. Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) input data rates may be increased by an order of magnitude, (2) test cases that require maximum memory are executed.

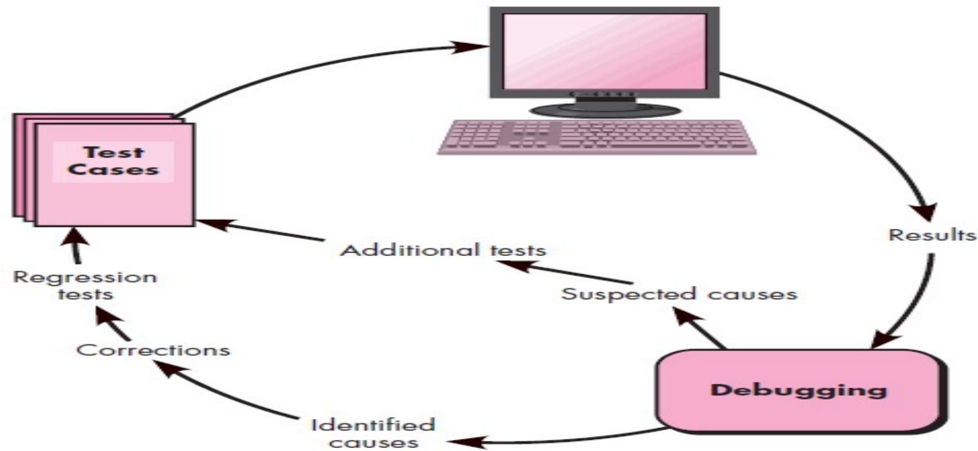
**Performance Testing** For real-time and embedded systems, software that does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process.

**Deployment Testing** In general, software must execute on a variety of platforms and under more than one operating system environment. Deployment testing, exercises the software in each environment in which it is to operate. It examines all installation procedures and specialized installation software that will be used by customers, and all documentation that will be used to introduce the software to end users.

**5) Debugging**

Debugging has one objective, to find and correct the cause of a software error or defect. Debugging is not testing but often occurs as a consequence of testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.

In many cases, the noncorresponding data are a symptom of an underlying cause. The debugging process attempts to match symptom with cause, thereby leading to error correction. The debugging

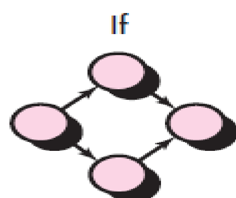



process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion. In general, three debugging strategies have been proposed (1) brute force, (2) backtracking, and (3) cause elimination. The **brute force** category of debugging is probably the most common method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Here, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. We hope that somewhere, we will a clue that can lead to the cause of an error.

**Backtracking** can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. As the number of source lines increases, the number of potential backward paths may become large. **Cause elimination**—data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the data are used to prove or disprove the hypothesis.

**6) White-box testing** using white-box testing, sometimes called glass-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

**i) Basis path testing** is a white-box testing technique. Here, a flow graph is created for the representation of control flow. Flow graph consists of nodes that represent one or more procedural statements. The arrows called edges that represent flow of control. An edge must terminate at a node. A flow graph should be drawn only when the logical structure of a component is complex. The flow graph allows you to trace program paths more readily. An example flow graph of "if" control flow is shown in Figure given below.



An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

How do you know how many paths to look for? cyclomatic complexity defines the number of independent paths and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once. Cyclomatic complexity can be the number of regions of the flow graph.

The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to do basis path testing.

1) Using the design or code as a foundation, draw a corresponding flow graph.
2) Determine the cyclomatic complexity of the resultant flow graph.
3) Determine a basis set of linearly independent paths.
4) Prepare test cases that will force execution of each path in the basis set.

**Graph matices**-A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing. A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

**ii) Control structure testing:** the basis path testing technique is one of techniques for control structure testing. Though it is simple and highly efficient, It is not sufficient in itself. Condition testing is a test-case design method that exercises the logical conditions contained in a program module. If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

**iii)Data flow testing method** selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

**iv) Loop testing** is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops. Simple loops. The following set of tests can be applied to simple

loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.

**7) Black-Box testing:**

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. This testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing attempts to find errors in the following categories: (1) incorrect functions, (2) interface errors, (3) errors in data structures, (4) behavior or performance errors, and (5) initialization and termination errors. Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing.

**Graph-Based Testing Methods**- software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered. To accomplish these steps, you begin by creating a graph—a collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behavior), and link weights that describe some characteristic of a link.

**Equivalence Partitioning** equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

**Boundary Value Analysis**

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values. Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

**Orthogonal Array Testing**

There are many applications, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. However, as the number of input values grows and the

number of discrete values for each data item increases, exhaustive testing becomes impractical. **Orthogonal array testing** can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component. When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. In the L9 orthogonal array, test cases are "dispersed uniformly throughout the test domain". The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases.

**8) Risk Management**

Risk analysis and management are actions that help a software team to understand and manage uncertainty. **A risk** is a potential problem-it might happen, it might not. 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. Risk always involves two characteristics: uncertainty-the risk may or may not happen; and loss-if the risk becomes a reality, unwanted consequences or losses will occur.

A **reactive strategy** monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. 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. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

**Software risks**- if project risks become real, it is likely that the project schedule will slip and that costs will increase. If a technical risk becomes a reality, implementation may become difficult or impossible. Business risks threaten the viability of the software to be built. Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment. Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks can and do occur, but they are extremely difficult to identify in advance.

**Risk identification** is a systematic attempt to specify threats to the project plan. Identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary. There are two types of risks: generic risks and 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 software that is to be built. One method for identifying risks is to

create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks. Some of the known and pridictable risk categories are product size—risks associated with the overall size of the software to be built or modified. Development environment—risks associated with the availability and quality of the tools to be used to build the product. Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

**Risk projection, also called risk estimation**, attempts to rate each risk in two ways (1) the probability that the risk is real and (2) the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

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.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

**Developing a Risk Table** a risk table provides you with a simple technique for risk projection. You begin by listing all risks in the first column of the table. This can be accomplished with the help of the risk item checklists. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of

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

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

risk probability begins to converge. The categories for each of the four risk components performance, support, cost, and schedule—are averaged to determine an overall impact value. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom.

**Risk Refinement:**

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more

detailed risks, each somewhat easier to mitigate, monitor, and manage. Refinement helps to isolate the underlying risks and might lead to easier analysis and response. One way to do this is to represent the risk in condition-transition-consequence (CTC) format.

**9) Mitigation, Monitoring, Management:**

All of the **risk analysis activities** have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning. If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for risk mitigation. For example, assume that high staff turnover is noted as a project risk. To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are: • Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay ). • Mitigate those causes that are under your control before the project starts.

As the project proceeds, **risk-monitoring** activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored.

**Risk management and contingency** planning assumes that mitigation efforts have failed and that the risk has become a reality. Example, the project is well under way and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team.

**The RMMM plan:**

The risk management steps can be organized into a separate risk mitigation, monitoring, and management plan (RMMM). 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. Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. **Risk mitigation** is a problem avoidance activity. **Risk monitoring** is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do 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.

## Process and Project Metrics ( UNIT- IV)

**Measurement** can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. It can be used by software engineers to help assess the quality of work products and to assist in tactical decision making as a project proceeds

**1) Software process and project metrics** are quantitative measures that enable you to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

**Process vs. Project metrics:**

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement. Project metrics enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go “critical,” (4) adjust work flow or tasks, and (5) evaluate the project team’s ability to control quality of software work products.

**Process metrics**

Referring to Figure 1 ( triangle inside circle) , process sits at the center of a triangle connecting three factors that have influence on software quality and organizational performance. The skill and motivation of people, the complexity of the product, the technology have a substantial impact on impact on quality and team performance. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

You can only measure the efficacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. You can also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, you might measure the effort and time spent performing the umbrella activities and the generic software engineering activities.

Grady argues that there are “private and public” uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis. Examples of private metrics include defect rates (by individual), defect rates (by

component), and errors found during development. Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during technical reviews, and lines of code or function points per component or function.

**Project Metrics**

Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress. Project metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. These are also used to assess product quality on an ongoing basis and modify the technical approach to improve quality.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

**2) Software Measurement:** software metrics can be categorized into direct measures and indirect measures. The direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities".

The software metrics domain can be partitioned into process, project, and product metrics and noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole.

**a) Size-Oriented Metrics:**

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. We can choose lines of code as a normalization value. A set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- $ per KLOC
- Pages of documentation per KLOC

Size-oriented metrics are not universally accepted as the best way to measure the software process.

**b) Function-Oriented Metrics:** Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.

**c) Reconciling LOC and FP Metrics**

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation, an historical baseline of information must be established.

**d) Object-Oriented Metrics**

Lorenz and Kidd suggest the following set of metrics for OO projects:

**Number of scenario scripts.** A scenario script (analogous to use cases) is a detailed sequence of steps that describe the interaction between the user and the application.

**Number of key classes.** Key classes are the "highly independent components" that are defined early in object-oriented analysis. The number of these is an indication of the amount of effort required to develop the software.

**Number of support classes.** Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (GUI) classes, database access and manipulation classes, and computation classes.

**Average number of support classes per key class.** If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified.

**Number of subsystems.** A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

**e) Use-Case–Oriented Metrics**

**Us**e cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP. Use cases describe (indirectly, at least) user-

visible functions and features that are basic requirements for a system. The use case is independent of programming language. In addition, the number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use cases can be created at vastly different levels of abstraction, there is no standard “size” for a use case. Researchers have suggested use-case points (UCPs) as a mechanism for estimating project effort and other characteristics. The UCP is a function of the number of actors and transactions implied by the use-case models and is analogous to the FP in some ways.

**3) Metrics for software Quality** Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Metrics for measuring the quality are given below:

**Defect Removal Efficiency.** A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities. When considered for a project as a whole, DRE is defined in the following manner:

DRE = E/(E+D), where E is the number of errors found before delivery of the software to the end user and D is the number of defects found after delivery. The ideal value for DRE is 1.

## Product Metrics

A key element of any engineering process is measurement. We can use measures to better understand the attributes of the models that you create and to assess the quality of the engineered products or systems that you build. Direct measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world. Because software measures and metrics are often indirect, they are open to debate.

**4) Metrics for the requirements model.**

**a) Function-Based Metrics.** The function point (FP) metric can be used effectively as a means for measuring the functionality delivered by a system. Using historical data, the FP metric can then be used to (1) estimate the cost required (2) predict the number of errors and (3) forecast the number of components and/or the number of projected source lines in the implemented system.

Function points are derived using an empirical relationship based on countable measures of software’s information domain and qualitative assessments of software complexity. Information domain values are defined in the following manner: ( Refer diagram for FPs computation)

| Information Domain Value | Count | | Weighting factor simple | average | complex | | |
|---|---|---|---|---|---|---|---|
| External Inputs (EIs) | | 3 | 3 | 4 | 6 | = | |
| External Outputs (EOs) | | 3 | 4 | 5 | 7 | = | |
| External Inquiries (EQs) | | 3 | 3 | 4 | 6 | = | |
| Internal Logical Files (ILFs) | | 3 | 7 | 10 | 15 | = | |
| External Interface Files (EIFs) | | 3 | 5 | 7 | 10 | = | |
| Count total → | | | | | | | |

**Number of external inputs (EIs).** Each one originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update internal logical files (ILFs).

**Number of external outputs (EOs).** Each one is derived data within the application that provides information to the user. Examples. reports, screens, error messages, etc.

N**umber of external inquiries (EQs).** Each one is defined as an online input that results in the generation of some immediate software response in the form of an online output.

**Number of internal logical files (ILFs).** Each one is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

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

To compute function points (FP), the following relationship is used:

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

**b) Metrics for Specification Quality.**

A list of characteristics that can be used to assess the quality of the requirements model and the corresponding requirements specification: specificity (lack of ambiguity), completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability. Each can be represented using one or more metrics.

$\mathbf{n_r}$ requirements in a specification, such that $n_r = n_f + n_{nf}$

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

To determine the **specificity** (lack of ambiguity) of requirements, which a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$Q_1 = n_{ui} / n_r$

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

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

$Q_2 = n_u / (n_i * n_s)$

where $n_u$ is the number of unique functional requirements, $n_i$ is the number of inputs defined or implied by the specification, and $n_s$ is the number of states specified. The $Q_2$ ratio measures the percentage of necessary functions that have been specified for a system.

**5) Metrics for the Design Model**

The design of complex software-based systems often proceeds with virtually no measurement. Design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence. Design metrics for computer software, like all other software metrics, are not perfect.

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. Card and Glass define three software design complexity measures: structural complexity, data complexity, and system complexity.

For hierarchical architectures (e.g., call-and-return architectures), structural complexity of a module i is defined in the following manner:

$S(i) = f^2_{out}(i)$

where $f_{out}(i)$ is the fan-out of module i.

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

$D(i) = v(i) / (f_{out}(i) + 1)$

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

specified as

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

**6) Metrics for Source code**

Halstead uses the following primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software.

These may be derived after code is generated or estimated once design is complete. The primitive measures are:

$n_1$ = number of distinct operators that appear in a program

$n_2$ = number of distinct operands that appear in a program

$N_1$ = total number of operator occurrences

$N_2$ = total number of operand occurrences

Halstead shows that length N can be estimated

$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$

and program volume may be defined

$V = N \log_2 (n_1 + n_2)$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program. Theoretically, a minimum volume must exist for a particular algorithm.

**7 ) Metrics for Testing**

**Architectural design metrics** provide information on the ease or difficulty associated with integration testing and the need for specialized testing software (e.g., stubs and drivers). Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with **high cyclomatic complexity** are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, you should expend above average effort to uncover errors in such modules before they are integrated in a system.

**a) Halstead Metrics Applied to Testing**

Testing effort can be estimated using metrics derived from Halstead measures. Using the definitions for program volume V and program level PL, Halstead effort e can be computed as

$PL = 1/ ((n_1/2) * (N_2/n_2))$

$e = V/PL$

**b) Metrics for Object-Oriented Testing**

Binder suggests a broad array of design metrics that have a direct influence on the “testability” of an OO system. The metrics consider aspects of encapsulation and inheritance.

**Lack of cohesion in methods (LCOM).** The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

**Percent public and protected (PAP).** It indicates the percentage of class attributes that are public or protected. High values for PAP increase the likelihood of side effects among classes.

**Public access to data members (PAD).** It indicates the number of classes (or methods) that can access another class’s attributes, a violation of encapsulation. High values for PAD lead to the potential for side effects among classes.

**Number of root classes (NOR).** This metric is a count of the distinct class hierarchies that are described in the design model.As NOR increases, testing effort also increases.

**Fan-in (FIN).** fan-in in the inheritance hierarchy is an indication of multiple inheritance. FIN > 1 indicates that a class inherits its attributes and operations from more than one root class. FIN > 1 should be avoided when possible.

**Number of children (NOC)** and depth of the inheritance tree (DIT). Superclass methods will have to be retested for each subclass.

**8) Metrics for Maintenance**

All of the software metrics introduced for requirements model, design, coding, testing in the above can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

Software maturity index (SMI) is a metric that provides an indication of the stability of a software product. The following information is determined:

$M_T$ = number of modules in the current release

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

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

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

The software maturity index (SMI) is computed in the following manner:

$SMI = (M_T - (F_a + F_c + F_d))/ M$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities.

**(UNIT-V)**

**Software Quality Assurance (SQA)**

If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time-to-market. It is important to define software quality at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

**1) A Software Quality Assurance Plan** is created to define a software team's SQA strategy. During modeling and coding, the primary SQA work product is the output of technical reviews. During testing, test plans and procedures are produced. Other work products associated with process improvement may also be generated.

Find errors before they become defects! That is, work to improve your defect removal efficiency, thereby reducing the amount of rework that your software team has to perform.

Software quality assurance (often called **quality management**) is an umbrella activity that is applied throughout the software process. It is a "planned and systematic pattern of actions" that are required to ensure high quality in software.

Software quality assurance (SQA) encompasses (1) an SQA process, (2) specific quality assurance and quality control tasks, (3) effective software engineering practice (methods and tools), (4) control of all software work products and the changes made to them, (5) a procedure to ensure compliance with software development standards, and (6) measurement and reporting mechanisms.

The people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors? Has software development been conducted according to preestablished standards? Have technical disciplines properly performed their roles as part of the SQA activity? The SQA group attempts to answer these.

**2) Software quality assurance** encompasses a broad range of **activities** that focus on the management of software quality.

**Standards.** The IEEE, ISO have produced a broad array of software engineering standards. The job of SQA is to ensure that standards that have been adopted are followed.

**Reviews and audits.** Technical reviews are a quality control activity

performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work.

**Testing.** Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted.

**Error/defect collection and analysis.** The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

**Change management.** Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.

**Education.** The SQA organization takes the lead in software process improvement and is a sponsor of educational programs.

**Security management.** With the increase in cyber crime and new government regulations regarding privacy, SQA ensures that appropriate process and technology are used to achieve software security.

**Safety.** SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

**Risk management.** the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

SQA works to ensure that software support activities (e.g., maintenance, help lines, documentation, and manuals) are conducted or produced with quality as a dominant concern.

**3) SQA tasks, goals, and metrics**

Software quality assurance is composed of a variety of tasks associated with two different groups of people—the software engineers who do technical work and an SQA group. Software engineers perform quality control activities by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

**SQA Tasks**

The SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis, and reporting. These actions are performed by an independent SQA group that:

**Prepares an SQA plan for a project.** The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software 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 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. 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. Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

**The SQA tasks(actions)** described above are performed to achieve a set of **goals**:

**Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. Example metrics- number UML models, number of UML errors.

**Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. Example metrics- number of components, number of patterns and existence of architectural model.

**Code quality.** Source code and related work products must conform to local coding standards and exhibit characteristics that will facilitate maintainability. Example metrics include variable naming conventions, percent internal comments, Cyclomatic complexity.

**Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. Example metrics- number of errors found, origin of error, actual vs. budgeted completion time.

**4) Statistical software quality assurance** It implies the following steps: 1. Information about software errors and defects is collected and categorized. 2. An attempt is made to trace each error and defect to its underlying cause. 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 errors and defects.

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!

This relatively simple concept represents an important step toward the creation of an adaptive software process in which changes are made to improve those elements of the process that introduce error.

**5) Six Sigma for Software Engineering** Six Sigma is the most widely used strategy for statistical quality assurance in industry today. It is “is a rigorous and disciplined methodology that uses data

and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes". The Six Sigma methodology defines three core steps:

• **Define** customer requirements and deliverables and project goals via well-defined methods of customer communication.

• **Measure** the existing process and its output to determine current quality performance (collect defect metrics).

• **Analyze** defect metrics and determine the vital few causes. If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

• Improve the process by eliminating the root causes of defects.

• Control the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

• **Design** the process to (1) avoid the root causes of defects and (2) to meet customer requirements.

• **Verify** that the process model will avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

**6) Software Reliability**

Software reliability can be measured directly and estimated using historical and developmental data. 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". To illustrate, program X is estimated to have a reliability of 0.999 over eight elapsed processing hours. In other words, if program X were to be executed 1000 times and require a total of eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 999 times. Failure is nonconformance to software requirements. the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures. all software failures can be traced to design or implementation problems rather than physical wear.

**Measures of Reliability and Availability**

a) If we consider a computer-based system, a simple measure of reliability is meantime-between-failure (MTBF):

MTBF = MTTF+ MTTR

where the MTTF and MTTR are mean-time-to-failure and mean-time-to-repair respectively.

b) An alternative measure of reliability is failures-in-time (FIT)—a statistical measure of how many failures a component will have over one billion hours of operation. Therefore, 1 FIT is equivalent to one failure in every billion hours of operation.

**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*100%)/(MTTF+MTTR)

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(SQA) 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, software design features can be specified that will either eliminate or control potential hazards. A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. Analysis techniques such as fault tree analysis, real-time logic, and Petri net models 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 these system-level hazardsare 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. 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.

**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. **Software safety** examines the ways in which failures result in conditions that can lead to a hazard.

**7) The ISO 9000 Quality standards:**

A quality assurance system implements  quality management. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations. These systems cover a wide variety of activities including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process.

ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

The requirements delineated by ISO 9001:2000 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of

quality records, internal quality audits, training, servicing, and statistical techniques. In order for a software organization to become registered to ISO 9001:2000, it must establish policies and procedures to address each of the requirements just noted and then be able to demonstrate that these policies and procedures are being followed.

**The following outline defines the basic elements of the ISO 9001:2000 standard.**

Establish the elements of a quality management system.

    Develop, implement, and improve the system.

Document the quality system.

    Describe the process.

    Produce an operational manual.

    Develop methods for controlling (updating) documents.

Support quality control and assurance.

    Promote the importance of quality among all stakeholders.

    Focus on customer satisfaction.

Establish review mechanisms for the quality management system.

    Identify review methods and feedback mechanisms.

    Define follow-up procedures.

Identify quality resources including personnel, training, and infrastructure elements.

    Establish control mechanisms for planning, customer requirements, technical activities

    For project monitoring and management

Define methods for remediation.

    Assess quality data and metrics.

**8) SQA Plan**

The SQA Plan provides a road map for software quality assurance. Developed by the SQA group, the plan serves as a template for SQA activities for each software project. The standard recommends a structure that identifies: (1) the purpose and scope of the plan, (2) a description of all software engineering work products (e.g., models, documents, source code), (3) all applicable standards and practices, (4) SQA actions and tasks (including reviews and audits) and their placement throughout the software process, (5) the tools and methods that support SQA actions and tasks, (6) software configuration management procedures, (7) methods for assembling, safeguarding, and maintaining all SQA-related records, and (8) organizational roles and responsibilities relative to product quality.

JNTU World

## PART – A (Short Answer Questions)

| S. No | Questions | Blooms Taxonomy Level | Course Outcome |
|---|---|---|---|
| | **UNIT – I** | | |
| 1 | **Explain** is legacy software? | Knowledge | 1 |
| 2 | **Demonstrate** all the applications of software? | Knowledge | 2 |
| 3 | **List** the types of software myths? | Knowledge | 2 |
| 4 | **Discuss** the architecture of layered technology? | Understand | 2 |
| 5 | **List** all the umbrella activities in process framework? | Understand | 2 |
| 6 | **Explain** is process pattern? | Knowledge | 2 |
| 7 | **List** the types of software models? | Understand | 2 |
| 8 | **List** the types other software process models? | Understand | 2 |
| 9 | **Explain** software component? explain its uses | Understand | 2 |
| 10 | **Explain** process assessment? | Knowledge | 2 |
| 11 | **List** the models in CMMI? | Knowledge | 2 |
| 12 | **Explain** the levels in continuous model in CMMI? | Understand | 2 |
| 13 | **Compare** between perspective and iterative process models? | Understand | 2 |
| 14 | **Explain** staged model in CMMI? | Knowledge | 2 |
| 15 | **Write the** other name of waterfall model and who invented waterfall model? | Understand | 2 |
| 16 | **Explain** Boehm model? | Understand | 2 |
| 17 | **List** the phases in unified process model?? | Understand | 2 |
| 18 | **List** the types of patterns? | Knowledge | 3 |
| 19 | **Explain** PSP and TSP? | Knowledge | 3 |
| 20 | **Explain** high speed adaptation model? | Understand | 3 |
| | **UNIT - II** | | |
| 1 | **Explain** the kinds of system requirements? | Knowledge | 3 |
| 2 | **Explain** functional requirement? | Knowledge | 3 |
| 3 | **Explain** nonfunctional requirement? | Understand | 3 |

Downloaded From JNTU World (http://www.alljntuworld.in)

| 4 | **Explain** domain requirements? | Understand | 3 |
|---|---|---|---|
| 5 | **List** kinds of nonfunctional requirements? | Knowledge | 3 |
| 6 | **Explain** example of functional requirement? | Understand | 3 |
| 7 | **Explain** user requirements in detail? | Understand | 3 |
| 8 | **Explain** system requirement in detail? | Understand | 4 |
| 9 | **Explain** interface and list out how many types of there and what are they? | Knowledge | 4 |
| 10 | **Explain** the term stake holder? | Knowledge | 4 |
| 11 | **Explain** requirements gathering?? | Knowledge | 4 |
| 12 | **Explain** requirement validation? | Understand | 5 |
| 13 | **Explain** requirement review? | Understand | 5 |
| 14 | **Explain** data dictionary? | Understand | 5 |
| 15 | **Discuss** data flow model? | Knowledge | 5 |
| 16 | **Explain** state machine model of a microwave oven? | Knowledge | 5 |
| 17 | **List** kinds of behavioral and object models? | Knowledge | 5 |
| 18 | **Design** class hierarchy for library by using in heritance model? | Knowledge | 5 |
| 19 | **Describe** ethnography? | Understand | 5 |
| 20 | **Explain** viewpoints and types of viewpoints? | Understand | 5 |
| | **UNIT - III** | | |
| 1 | **Explain** why design is important in design engineering? | Knowledge | 4 |
| 2 | **Discuss** analysis and design model? | Understand | 4 |
| 3 | **Describe** quality attributes and its guidelines? | Understand | 5 |
| 4 | **List** the design concepts? | Knowledge | 5 |
| 5 | **Justify** the importance of refactoring? | Understand | 5 |
| 6 | **Discuss** on low coupling? | Understand | 5 |
| 7 | **Define** software architecture with its importance? | Understand | 5 |
| 8 | **Explain** taxonomy of architectural styles? | Knowledge | 5 |
| 9 | **Write** short notes on architecture patterns? | Knowledge | 6 |
| 10 | **Compare** function oriented and object oriented design? | Understand | 6 |
| 11 | **Define top**-down and bottom-up design model? | Knowledge | 6 |
| 12 | **Write** short notes on coupling? | Knowledge | 6 |
| 13 | **List** out the steps for conducting component level design? | Knowledge | 6 |
| 14 | **Write** short notes on cohesion? | Knowledge | 6 |
| 15 | **Design** the class based components? | Understand | 6 |
| 16 | **List** out the golden rules for interface design? | Understand | 7 |
| 17 | **Write** short notes on interface design steps? | Knowledge | 7 |
| 18 | **Describe** design evaluation? | Knowledge | 7 |
| 19 | **List** out all the design issues? | Understand | 7 |
| 20 | **Explain** process in user interface design? | Understand | 7 |
| | **UNIT - IV** | | |
| 1 | **Compare** verification and validation? | Knowledge | 6 |
| 2 | **Write** short notes on unit testing? | Knowledge | 6 |
| 3 | **Describe** smoke testing? | Knowledge | 6 |
| 4 | **List** out the steps for bottom-up integration? | Knowledge | 6 |
| 5 | **List** out the steps for top-down integration? | Understand | 7 |
| 6 | **Write** short note on integration testing? | Understand | 7 |
| 7 | **Compare** Quality assurance vs. Quality Control? | Knowledge | 7 |
| 8 | **Define** CASE tools? | Knowledge | 7 |
| 9 | **Write** short notes on validation testing? | Knowledge | 7 |
| 10 | **Explain** art of debugging? | Understand | 7 |
| 11 | **Describe** regression testing? | Knowledge | 9 |
| 12 | **List** out the steps for integration step documentation? | Knowledge | 9 |
| 13 | **Describe** performance testing? | Knowledge | 9 |
| 14 | **Write** short notes on glass box testing? | Knowledge | 9 |
| 15 | **Explain** behavioral testing? | Understand | 9 |
| 16 | **List** the quality factors of McCall’s? | Understand | 9 |
| 17 | **List** the quality factors of ISO 9126? | Knowledge | 9 |
| 18 | **Define** the following terms measures, metrics, and indicators? | Understand | 9 |
| 19 | **Write** short notes on product metric land scrape? | Understand | 9 |
| 20 | **List** out the metrics for analysis model? | Understand | 9 |
| | **UNIT - V** | | |

JNTU World

| | | | |
|---|---|---|---|
| 1 | **Define** reactive and proactive risk strategies? | Knowledge | 8 |
| 2 | **List** out the generic subcategories of predictable risks? | Understand | 8 |
| 3 | **Define** risk components? | Understand | 8 |
| 4 | **List** out the conditions for risk refinement? | Knowledge | 9 |
| 5 | **Write about** quality concepts? | Understand | 9 |
| 6 | **Write** short notes on formal technical reviews? | Understand | 9 |
| 7 | **List** out review guidelines?? | Understand | 9 |
| 8 | **Describe** six sigma for software? | Knowledge | 9 |
| 9 | **Write** about the classification of case tool? | Knowledge | 9 |
| 10 | **Write** a short notes on ISO 9000 quality standards? | Understand | 9 |
| 11 | **Write** the formulae for measures of reliability and availability? | Knowledge | 9 |
| 12 | **Explain** about software cost estimation? | Knowledge | 10 |
| 13 | **Write** short note on the various estimation techniques? | Knowledge | 10 |
| 14 | **Define** software risks and what are the types of software risks? | Knowledge | 10 |
| 15 | **Describe** risk components and drivers? | Understand | 10 |
| 16 | **Write** the purpose of timeline chart? | Understand | 10 |
| 17 | **Expand** RMMM in RMMM plan? | Knowledge | 10 |
| 18 | **Define** software reliability? | Understand | 10 |
| 19 | **Define** quality and quality control in quality management? | Understand | 11 |
| 20 | **Write** short notes on risk identification? | Understand | 11 |

## PART – B (Long Answer Questions)

| S. No | Questions | Blooms Taxonomy Level | Course Outcome |
|---|---|---|---|
| | **UNIT – I** | | |
| 1 | **Explain** the evolving role of software? | Knowledge | 1 |
| 2 | **Define** software and explain the various characteristics of software? | Knowledge | 2 |
| 3 | **Describe** "Software myth"? Discuss on various types of software myths and the true aspects of these myths? | Knowledge | 2 |
| 4 | **Explain** software Engineering? Explain the software engineering layers? | Understand | 2 |
| 5 | **Explain** in detail the capability Maturity Model Integration (CMMI)? | Understand | 2 |
| 6 | **Describe** with the help of the diagram discuss in detail waterfall model. Give certain reasons for its failure? | Knowledge | 2 |
| 7 | **Explain** briefly on (a) the incremental model (b) The RAD Model? | Understand | 2 |
| 8 | **Explain** the Spiral model in detail? | Understand | 2 |
| 9 | **Describe** With the help of the diagram explain the concurrent development model? | Understand | 2 |
| 10 | **Explain** unified process? Elaborate on the unified process work products? | Knowledge | 3 |
| 11 | **Explain** specialized process models? | Knowledge | 3 |
| 12 | **Explain** different software applications? | Knowledge | 3 |
| 13 | **Explain** the paradigms do you think would be most effective? Why? | Understand | 3 |
| 14 | **Explain** product and process are related? | Understand | 3 |
| 15 | **Explain** personal and team process models? | Understand | 3 |
| 16 | **Explain** process frame work activities? | Knowledge | 3 |
| 17 | **Explain** the purpose of process assessment? | Knowledge | 3 |
| 18 | **Explain** changing nature of software in detail? | Knowledge | 3 |
| 19 | **Explain** and contrast perspective process models and iterative process models? | Understand | 3 |
| 20 | **Explain** about the evolutionary process models? | Understand | 3 |
| | **UNIT – II** | | |
| 1 | **Write** short notes on user requirements. What are requirements? | Knowledge | 3 |
| 2 | **Compare** functional requirements with nonfunctional requirements? | Knowledge | 3 |
| 3 | **Discuss** system requirements in a detail manner? | Understand | 3 |
| 4 | **Explain** requirement engineering process? | Understand | 3 |
| 5 | **Discuss** briefly how requirement validation is done? | Knowledge | 3 |
| 6 | **Discuss** your knowledge of how an ATM is used; develop a set of use-cases that could serve as a basis for understanding the requirements for an ATM system? | Understand | 3 |

Downloaded From JNTU World (http://www.alljntuworld.in)

| 7 | **Describe** four types of non-functional requirements that may be placed on a system. Give examples of each of these types of requirement? | Understand | 3 |
|---|---|---|---|
| 8 | **Explain** how requirements are managed in software project management? | Understand | 4 |
| 9 | **Explain** context models? | Knowledge | 4 |
| 10 | **Explain** Behavioral models? | Knowledge | 4 |
| 11 | **Explain** Data models? | Knowledge | 4 |
| 12 | **Explain** Object models? | Understand | 4 |
| 13 | **Explain** in which circumstances would you recommend using structured methods for system development? | Understand | 4 |
| 14 | **Explain** SRS document and explain along with its contents? | Understand | 4 |
| 15 | **Explain** interface specification in detail? | Knowledge | 4 |
| 16 | **Discuss** how requirements are felicitated and validated in software project? | Knowledge | 4 |
| 17 | **Discuss** how feasibility studies are important in requirement engineering process? | Knowledge | 4 |
| 18 | **Demonstrate** class hierarchy for library by using interface specification? | Understand | 4 |
| 19 | **Explain** inheritance model? | Understand | 4 |
| 20 | **Explain** state machine model with a suitable example? | Understand | 4 |
| | **UNIT - III** | | |
| 1 | **Explain** a two level process? Why should system design be finished before the detailed design, rather starting the detailed design after the requirements specification? Explain with the help of a suitable example | Knowledge | 4 |
| 2 | **Discuss** briefly the following fundamental concepts of software design:<br>i) Abstraction<br>ii) Modularity<br>iii) Information hiding | Understand | 4 |
| 3 | **Explain** briefly the following:<br>i) Coupling between the modules,<br>ii) The internal Cohesion of a module | Understand | 5 |
| 4 | **Discuss** the fundamental principles of structured design. Write notes on transform analysis? | Knowledge | 5 |
| 5 | **Explain** software architecture in a detail manner? | Understand | 5 |
| 6 | **Explain** software design? Explain data flow oriented design? | Understand | 5 |
| 7 | **Explain** the goals of the user interface design? | Understand | 5 |
| 8 | **Discuss** briefly about the golden rules for the user interface design? | Knowledge | 5 |
| 9 | **Discuss** interface design steps in a brief manner? | Knowledge | 6 |
| 10 | **Explain** how the design is evaluated? | Understand | 6 |
| 11 | **Explain** design processing along with its quality? | Knowledge | 6 |
| 12 | **Explain** the design concepts in software engineering? | Understand | 6 |
| 13 | **Explain** pattern based software design in a detail manner? | Understand | 6 |
| 14 | **Elaborate** model for the design? | Understand | 6 |
| 15 | **Discuss** architectural styles and patterns? | Knowledge | 6 |
| 16 | **Explain** with a neat diagram of architectural design? | Knowledge | 6 |
| 17 | **Elaborate** modeling component level design? | Knowledge | 6 |
| 18 | **Describe** mapping data flow into software architecture? | Understand | 6 |
| 19 | **Explain** the guide lines of component level design? | Understand | 6 |
| 20 | **Describe** the way of conducting a component level design? | Understand | 6 |
| | **UNIT- IV** | | |
| 1 | **Explain** about the importance of test strategies for conventional software? | Knowledge | 6 |
| 2 | **Discuss** black box testing in a detailed view? | Apply | 6 |
| 3 | **Compare** black box testing with white box testing? | Apply | 6 |
| 4 | **Compare** validation testing and system testing? | Knowledge | 6 |
| 5 | **Discuss** software quality factors? Discuss their relative importance? | Understand | 7 |
| 6 | **Discuss** an overview of quality metrics? | Understand | 7 |
| 7 | **Explain** should we perform the Validation test – the software developer or the software user? Justify your answer? | Apply | 7 |
| 8 | **Explain** about Product metrics? | Knowledge | 7 |

JNTU World

| 9 | **Explain** about Metrics for maintenance? | Knowledge | 7 |
|---|---|---|---|
| 10 | **Explain** in detail about Software Measurement? | Understand | 7 |
| 11 | **Explain** about Metrics for software quality? | Knowledge | 7 |
| 12 | **Explain** strategic approach to software testing | Understand | 7 |
| 13 | **Describe** test strategies for conventional software | Understand | 7 |
| 14 | **Describe** validation testing | Understand | 7 |
| 15 | **Write** a long notes on system testing | Knowledge | 7 |
| 16 | **Demonstrate** art of debugging | Knowledge | 7 |
| 17 | **Discuss** a framework for product metrics | Knowledge | 7 |
| 18 | **Demonstrate** metrics for analysis model | Understand | 7 |
| 19 | **List** the metrics for the design model | Understand | 7 |
| 20 | **Describe** metrics for source code and for testing | Understand | 7 |
| | **UNIT - V** | | |
| 1 | **Explain** about software risks? | Knowledge | 8 |
| 2 | **Elaborate** the concepts of Risk management Reactive vs Proactive Risk strategies? | Understand | 8 |
| 3 | **Explain** about RMMM Plan? | Understand | 8 |
| 4 | **Explain** about Quality concepts? | Knowledge | 9 |
| 5 | **Explain** software quality assurance? | Understand | 9 |
| 6 | **Explain** about formal technical reviews? | Understand | 9 |
| 7 | **Explain** in detail ISO 9000 quality standards? | Understand | 9 |
| 8 | **Discuss** risk refinement? | Knowledge | 9 |
| 9 | **Compare** reactive with proactive risk strategies? | Knowledge | 9 |
| 10 | **Discuss** software reliability? | Understand | 9 |
| 11 | **Briefly** explain about formal approaches to SQA? | Knowledge | 9 |
| 12 | **Demonstrate** statistical SQA? | Understand | 9 |
| 13 | **Define** software reliability along with its terms? | Understand | 9 |
| 14 | **Explain** risk projection in detail? | Understand | 9 |
| 15 | **Explain** seven principals of risk management? | Knowledge | 9 |
| 16 | **Explain** software reviews in brief? | Knowledge | 9 |
| 17 | **Explain** six sigma for software engineering? | Knowledge | 9 |
| 18 | **Explain** quality management with their terms? | Understand | 9 |
| 19 | **Demonstrate** risk identification? | Understand | 9 |
| 20 | **Describe** developing a risk table? | Understand | 9 |

## PART – C (Problem Solving and Critical Thinking Questions)

| S. No | Questions | Blooms Taxonomy Level | Course Outcome |
|---|---|---|---|
| | **UNIT – I** | | |
| 1 | **Describe** the law of conservation of familiarity in your own words? | Knowledge | 1 |
| 2 | **Suggest** a few ways to build software to stop deterioration due to change? | Knowledge | 1 |
| 3 | **Try** to develop a task set for the communication activity? | Apply | 2 |
| 4 | **What** is the purpose of process assessment? Why has SPICE been developed as a standard for process assessment? | Knowledge | 2 |
| 5 | **Discuss** the meaning of "cross-cutting concerns" in your words? | Knowledge | 2 |
| | **UNIT – II** | | |
| 1 | **Identify** and briefly describe four types of requirements that may be defined for computer based system? | Knowledge | 3 |
| 2 | **List** out plausible user requirements for the following functions<br>a) Cash dispensing function in a bank ATM?<br>b) Spelling check and correcting function in a word processor? | Knowledge | 3 |
| 3 | **Suggest** how an engineer responsible for drawing up a system requirements specification might keep track of the relationship between functional and non- functional requirements? | Knowledge | 4 |
| 4 | **Suggest** who might be stakeholders in a university student record system. Explain why it is almost inevitable that the requirements of different stakeholders will conflict in some way? | Knowledge | 4 |
| 5 | **Explain** who should be involved in requirements review? draw a | Apply | 4 |

JNTU World

| | | | |
|---|---|---|---|
| | process model showing how a requirements review might be organized? | | |
| | **UNIT – III** | | |
| 1 | **State** how do we assess quality of a software design? | Knowledge | 5 |
| 2 | **Suggest** a design pattern that you encounter in a category of everyday things? | Apply | 5 |
| 3 | **Provide** examples of three data abstractions and the procedural abstractions that can be used to manipulate them? | Apply | 5 |
| 4 | **Explain** the difference between a data base that services one or more conventional business applications and data warehouse? | Knowledge | 5 |
| 5 | **Demonstrate** the architecture of a house or building as a metaphor, draw comparison with software architecture. How are the disciplines of classical architecture and software architecture similar? How do they differ? | Apply | 5 |
| | **UNIT – IV** | | |
| 1 | **Provide** a few examples that illustrate why response time variability can be an issue? | Knowledge | 6 |
| 2 | **Develop** two additional design principles "place the user in control"? | Apply | 6 |
| 3 | **Develop** two additional design principles "make the interface consistent"? | Apply | 7 |
| 4 | **Develop** a complete test strategy for the safe home system. Document it in a test specification. | Apply | 7 |
| 5 | **Provide** examples for unit testing? | Apply | 7 |
| | **UNIT – V** | | |
| 1 | **Quality** and reliability are related concepts but are fundamentally different in number of ways. Discuss them? | Apply | 8 |
| 2 | **You** have been given the responsibility for improving quality of software across your organization. What is the first thing that you should do? What's next? | Apply | 8 |
| 3 | **Some** people argue that an FTR should assess programming style as well as correctness is this a good idea? Why? | Apply | 8 |
| 4 | **Is** it possible to assess the quality of software if the customer keeps changing what it is supposed to do? | Apply | 9 |
| 5 | **Create** a risk table for the project that if you are the project manager for a major software company. you have been asked to lead a team that's developing "next generation "word- processing software? | Apply | 9 |

JNTU World

Downloaded From JNTU World (http://www.alljntuworld.in)

**SE Assignment-1.**

**Write your assignment in NOT less than 4 pages ( use pencil for diagrams)**

**For 501-515 Roll numbers:**

Explain unified process (UP) model with neat sketch.
Define process assessment. Explain CMMI for process assessemnt..

**Roll Numbers 516-530:**

Define functional, non-functional requirements.
Give differences between user requirements and system requirements.
Requirements elicitation in detail.

**Roll Numbers 530-545:**

Compare various software process models. Describe spiral and incremental models with diagrams.
Changing nature of software.

**Roll Numbers 546-560 and LE Students:**

Describe software process framework.
Requirements validation process ( checks to perfom validation and validation techniques).

# MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

Chaitanya Bharati P.O., Gandipet, Hyderabad- 500 075

## III B.Tech, I - SEM, I - MID DESCRIPTIVE EXAM - A.Y: 2021-22

| | | | |
|---|---|---|---|
| **SUBJECT** | **: Software Engineering** | **Duration** | **: 1 Hr** |
| **Date & Session** | **: 08/11/2021 (AN)** | **Maximum Marks** | **: 10 Marks** |
| **Year, Class & Section** | **: III / IV B. Tech CSE & IT** | **Roll Number** | **:** |

| | **NOTE:Answer any TWO questions. All questions carry equal marks.** | Course Outcomes | Marks |
|---|---|---|---|
| 1 | a) What is myth? Explain different software myths. | CO1 | 3 M |
| | b) Justify why Software Engineering is a layered technology. | CO1 | 2 M |
| 2 | a) Give brief note on CMMMI. | CO1 | 2 M |
| | b) Compare and contrast different proces models. | CO1 | 3 M |
| 3 | a) Differentiate User and System requirements. | CO2 | 2 M |
| | b) Enumerate and Explain the structure of SRS document. | CO2 | 3 M |
| 4 | a) Explain the Requirements discovery methods. | CO2 | 3 M |
| | b)Explain the following Design Concepts: *Abstraction, Information hiding, Modularity, Refactoring.* | CO3 | 2 M |

**Paper set By: Department of CSE & IT**

# MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

## III B.Tech. I Sem, I MID –OBJECTIVE EXAM, A.Y: 2021-22

10

**Subject: Software Engineering** **Hall Ticket. No: _____________**

**Year, Branch & Section: III/IV, CSE & IT** **Date and Session: 08/11/2021(AN)**

**Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 10**

1. Which among the following is Umbrella activity? [ ]

   A) Planning B) Construction C) Risk management D) Design

2. Which process model is suitable when requirements are fixed? [ ]

   A) Incremental B) Waterfall C) Prototype D) Spiral

3. In incremental model, the first increment is __________________ [ ]

   A) Increment B) Core product C) Failure D) Plan

4. In_________ model, multiple software teams works in parallel on different s/w functions. [ ]

   A) Waterfall B Unified C) Personal D) RAD

5. Quick Plan and Quick Design activities leads to construction of_____________ [ ]

   A) Prototype B) Code C) Iteration D) Increment

6. Which of the following is not an External requirement? [ ]

   A) Ethical B) Efficiency C) Legislative D) Safety/ Privacy

7. Which of the following OOPs principle enables code reusability [ ]

   A) Aggregation B) Association C) Inheritance D) Polymorphism

8. In__, Social Scientist spends considerable time observing & analyzing how people actually work [ ]

   A) Interviewing B) Ethnography C) Scenario D) Viewpoints

9. Which framework activity produces feedback report_______________ [ ]

   A) Communication B) Planning C) Modeling D) Deployment

10. ___________ Requirements provide detailed description of services from the stakeholder. [ ]

    A) User B) System C) Domain D) Non-functional

11. Software doesn't __________, but it does deteriorate.
12. The bedrock that supports Software Engineering________________.
13. _________________is proven solutions to the process-related problem.
14. Waterfall model is also called as________________________
15. _____________ Model can be used to apply throughout the entire life cycle of application.
16. CASE stands for _______________________________.
17. The process of finding out, Analyzing, Documenting and Checking the services and constraints is called as______________________.
18. _________________ assesses if the system is useful to the business or not.
19. Requirements Elicitation is also called as_________________________.
20. _____________ is an iterative process of translating requirements into blueprint of the system.

# MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

Chaitanya Bharati P.O., Gandipet, Hyderabad- 500 075

## III B.Tech, I - SEM, II - MID DESCRIPTIVE EXAM - A.Y: 2021-2022

| | | | |
|---|---|---|---|
| **SUBJECT** | **: Software Engineering** | **Duration** | **: 1 Hr** |
| **Date & Session** | **: 10/01/2022 (AN)** | **Maximum Marks** | **: 10 Marks** |
| **Year, Class & Section** | **: III / IV B. Tech CSE & IT** | **Roll Number** | **:** |

| | **NOTE:Answer any TWO questions. All questions carry equal marks.** | Course Outcomes | Marks |
|---|---|---|---|
| 1 | a) Discuss various Architectural Styles with neat sketch. | CO3 | 3 M |
| | b) Draw a class diagram for a Web-Based Food- Ordering system such as “Swiggy”. | CO3 | 2 M |
| 2 | a) What is Debugging? Explain the Product Metrics for Source Code, Testing. | CO4 | 2 M |
| | b) Elaborate any three testing Strategies for Conventional Software. | CO4 | 3 M |
| 3 | a) Distinguish White-box and Block-box testing. Explain the techniques of White-box testing. | CO4 | 3M |
| | b) Provide the format of Risk Information Sheet. | CO5 | 2 M |
| 4 | a) Explain Formal Technical Reviews. | CO5 | 3 M |
| | b) List and Explain various Software Risks. | CO5 | 2 M |

**Paper set By: Department of CSE & IT**

# MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

**III B.Tech. I Sem, II MID –OBJECTIVE EXAM, A.Y: 2021-2022**

10

**Subject: Software Engineering** **Hall Ticket. No: _____________**

**Year, Branch & Section: _______________ Date and Session: 10/01/2022 & AN**

**Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks:10**

1. The main purpose of integration testing is to remove____________ [ ]
   a)Design errors b) Analysis errors c) Procedure errors d) Interface errors
2. A _____________ is a modular building block for computer software. [ ]
   a) Component b) System c) Device d) None of these
3. Risk Table does not include_____________ [ ]
   a) Risk b) Probability c) Impact d) Testing
4. What is MTTF? [ ]
   a) Max Time To Failure b) Mean Time To Failure c) Min Time To Failure d) Median Time To Failure
5. ___________ is a characteristic or attribute of something. [ ]
   a)Error b) Bug c) Defect d) Quality
6. According to Pareto's principle, x% of defects can be traced to y% of all causes. What is the ratio? [ ]
   a) 60, 40 b) 70, 30 c) 80, 20 d) No such principle exists
7. The primary objective of Formal Technical Reviews is to find _________ during the process so that they do not become defects after release of the software. [ ]
   a) Errors b) Equivalent faults c) Failure cause d) None of the mentioned
8. ________ Risk strategy does nothing about risks until something goes wrong. [ ]
   a) Reactive b) Pro-active c) Both a &b d)Active
9. _________is the probability of failure-free operation of a computer program in specified environment for a specified time. [ ]
   a) Correctness b) Maintainability c) Integrity d) Reliability
10. _______ Test is conducted by the end users. [ ]
    a) Alpha Testing b) System Testing c) Beta Testing d) Unit Testing
11. ________________________ focuses on testing the smallest unit of the software.
12. _______________________is the re-execution of some subset of tests to ensure that changes have not propagated.
13. ____________________ is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure and implementation details.
14. ________________________is a part-of –relationship.
15. _______________________is the process of removal of errors.
16. RMMM stands for _________________________________________________________.
17. Building blocks of UML are________________, __________________, & _____________.
18. MTBF =__________+____________.
19. The FTR is actually a class of reviews that includes__________________ and _______________.
20. McCall's quality factors are____________, ______________, _______________.

Paper set By: Department of CSE & IT

## Question Bank

### UNIT-I

1. Explain software Myths in detail.

2. Explain software engineering as a layered Technology with neat digram.

3. Explain software process framework and umbrella activities with neat figure.

4. Describe CMMI levels and its significance.

5. Evolving role of software (or) Dual role of software.

6. Process models: Spiral model, Incremental model, Unified process.

**UNIT-II**

1. Distinguish between functional and Non-funtional requirements.

2. Discuss the structure of Software Requirements Specification (SRS) document and its chapters in detail.

3. Explain requirements engineering activities in detail with spiral digram.

4. Requirements discovery (or) Elicitation approaches: interviewing, usecases, scenarios, ethnography.

5. Models may be asked for any E-commerce application (ex. Amazon, Swiggy etc. )
Know graphical symbols involved in the following Software system models:

Context models ( context diagram);
Interation models ( Use-case, sequence diagrams);
Structural models ( class-diagram, component diagram).
Behavioral models – activity, state diagrams.
Object models - see Pressman text-book.

**Unit-IV**

1. Discuss project metrics for all 5 stages of software development such as requiments gathering, design, coding, testing, maintenance.

2. Art of debugging with neat diagram.

3. Black-box testing strategies with diagrams .

4. Testing strategies for conventtional software with neat spiral diagram.

**Unit-V**

1. Formal Technical Reviews (FTR) in detail.
2. Software Quality Assurance (SQA) activities.
3. Proactive vs. Reactive risk strategies.
4. RMMM- risk monitoring mitigation and managment in detail.
5. Risk information sheet with example.( RMMM plan)
6. Risk table. ( Risk identification, risk projection. )