KARPAGAM ACADEMY OF HIGHER EDUCATION

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE: 17CTU402** **SYLLABUS** **SEMESTER : IV**
**BATCH (2017-2020)**

**Semester – IV**

**17CTU402 SOFTWARE ENGINEERING** **4H – 4C**

**Instruction Hours / week:L: 4 T: 0 P: 0** **Marks:**Int :**40** Ext : **60** Total: **100**

**SCOPE**

The graduates of the software engineering program shall be able to apply proper theoretical, technical, and practical knowledge of software requirements, analysis, design, implementation, verification and validation, and documentation. This course enables the students to resolve conflicting project objectives considering viable tradeoffs within limitations of cost, time, knowledge, existing systems, and organizations.

**OBJECTIVES**

- Apply their knowledge of mathematics, sciences, and computer science to the modeling, analysis, and measurement of software artifacts.
- Work effectively as leader/member of a development team to deliver quality software artifacts.
- Analyze, specify and document software requirements for a software system.
- Implement a given software design using sound development practices.
- Verify, validate, assess and assure the quality of software artifacts.
- Design, select and apply the most appropriate software engineering process for a given project, plan for a software project, identify its scope and risks, and estimate its cost and time.
- Express and understand the importance of negotiation, effective work habits, leadership, and good communication with stakeholders, in written and oral forms, in a typical software development environment.

**UNIT-I**

**Introduction:** The Evolving Role of Software, Software Characteristics, Changing Nature of Software, Software Engineering as a Layered Technology, Software Process Framework, Framework and Umbrella Activities, Process Models, Capability Maturity Model Integration (CMMI).

**UNIT-II**

**Requirement Analysis;** Initiating Requirement EngineeringProcess- Requirement Analysis and Modeling Techniques- FlowOriented Modeling- Need for SRS- Characteristics and Components of SRS- Software Project Management: Estimation in Project Planning Process, Project Scheduling.

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE: 17CTU402** **SYLLABUS** **SEMESTER : IV**
**BATCH (2017-2020)**

**UNIT-III**

Risk Management: Software Risks, Risk Identification Risk Projection and Risk Refinement, RMMM plan, **QualityManagement-** Quality Concepts, Software Quality Assurance, Software Reviews, Metrics for Process and Projects

**UNIT-IV**

Design Engineering-Design Concepts, Architectural Design Elements, Software Architecture,Data Design at the Architectural Level and Component Level, Mapping of Data Flow into Software Architecture, Modeling Component Level Design

**UNIT-V**

**Testing Strategies & Tactics:** Software Testing Fundamentals, Strategic Approach to Software Testing, Test Strategies for Conventional Software, Validation Testing, System testing Black-Box Testing, White-Box Testing and their type, Basis Path Testing

**Suggested Readings**


1. R.S. Pressman, (2009). Software Engineering: A Practitioner‘s Approach (7th ed.). McGraw-Hill.
2. P.Jalote (2008). An Integrated Approach to Software Engineering (2nded.). New Age International Publishers.
3. K.K. Aggarwal andY.Singh (2008). Software Engineering ( 2nd ed.). New Age International Publishers.
4. Sommerville (2006). Software Engineering (8thed.). Addison Wesley.
5. D.Bell (2005). Software Engineering for Students (4thed.) Addison-Wesley.
6. R.Mall (2004). Fundamentals of SoftwareEngineering(2nd ed.). Prentice-Hall ofIndia.


**WEB SITES**

1. http://en.wikipedia.org/wiki/Software_engineering
2. http://www.onesmartclick.com/engineering/software-engineering.html
3. http://www.CC.gatech.edu/classes/AY2000/cs3802_fall/

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Pollachi Main Road, Eachanari Post, Coimbatore - 641021
(For the candidates admitted from 2016 onwards)
**DEPARTMENT OF CS, CA & IT**

STAFFNAME: **Dr.J.RAJESWARI** SUB.CODE: **17CTU402**
SUBJECT NAME: **SOFTWARE ENGINEERING** CLASS:**II B.Sc.(CT)**
SEMESTER: **IV**

| S.No | Lecture Duration Period | Topics to be Covered | Support Material/ Page Nos |
|---|---|---|---|
| | | **UNIT-I** | |
| 1 | 1 | Introduction to Software Engineering<br>TheEvolvingrole of Software<br>➢ SoftwareCharacteristics | S2:34-39<br><br>S2: 45-47 |
| 2 | 1 | ChangingNatureof software<br>➢ WebApps<br>➢ Mobile Applications<br>➢ Cloud computing<br>➢ Product line software<br>A Generic View of Process<br>➢ SoftwareEngineering-Layered Technology | S1:9-11<br><br><br><br>S2:52-54 |
| 3 | 1 | Softwareprocess<br>Framework Umbrella Activities | S1: 16-17<br>S1:18 |
| 4 | 1 | Process Models<br>➢ Prescriptive Models<br>➢ Waterfall Model<br>Incremental Process Models<br>➢ Incremental Model<br>➢ The RADModel | S1:41, W1,W2<br>S3:9-11<br><br>S1:43, W1 |

| | | | |
|---|---|---|---|
| 5 | 1 | EvolutionaryProcess Models<br>➢ Prototyping<br>➢ The Spiral Model | S1:45,W1<br>S2:85-88,138-153 |
| 6 | 1<br>1 | Specialized Process Models<br>➢ Component Based Development<br>➢ The Formal Methods Model<br>➢ Aspect Oriented Software Development | S1:52, W1<br><br>S2:59-61 |
| 7 | 1 | ➢ CapabilityMaturity ModelIntegration (CMMI) | S2:62-63 |
| 8 | 1 | Recapitulation and Discussion of important questions | |
| | **Total No of Hours Planned For UnitI** | | **8 Hours** |
| | | **UNIT-II** | |
| 1 | 1 | **RequirementAnalysis**<br>Requirement EngineeringTasks<br>Initiatingrequiremente ngineeringprocess<br>➢ Elicitingrequirements | S2: 176-181<br><br>S2:182:191 |
| 2 | 1 | Requirement Analysis Techniques<br>➢ Requirement Analysis<br>➢ Overall Objective and Philosophy<br>➢ Analysis Rules of Thumb | S2:207-211<br>S5:241-244 |
| 3 | 1 | Requirement ModelingApproaches<br>Data ModelingConcepts<br>➢ Data Objects<br>➢ Data Attributes<br>➢ Relationships | S2:211-215 |
| 4 | 1 | Flow Oriented Modeling<br>➢ CreatingData FlowModel<br>➢ Creating a Control FlowModel<br>➢ Need for SRS<br>➢ Characteristics and components of SRS | S2:226-229<br><br>S5: 101-103<br><br>S2:229-230<br>W2,W5 |
| 5 | 1 | SoftwareProject Management | S1:684 |

| | | | |
|---|---|---|---|
| 6 | 1 | Estimation in project planning process | S1:729 |
| 7 | 1 | Project Scheduling | S1:759 |
| 8 | 1 | Recapitulation and Discussion of important questions | |
| | **Total No of Hours Planned For UnitII** | | **8 Hours** |
| | | **UNIT-III** | |
| 1 | 1 | Risk Management<br>➢ SoftwareRisk | S1:778 |
| 2 | 1 | ➢ RiskIdentification<br>❖ Assessing Overall Project Risk<br>❖ Risk Components and Drivers | S1:780-782 |
| 3 | 1 | ➢ Risk Projection<br>❖ DevelopingaRisk Table<br>❖ AssessingRisk Impact<br>➢ Risk Refinement | S1:783-785<br><br><br><br>S1:787 |
| 4 | 1 | ➢ RMMM Plan<br><br>QualityManagement<br>QualityConcepts | S1:790<br><br>S1:287-289<br>S1:412-418 |
| 5 | 1 | SoftwareQualityAssurance | S1:448 |
| 6 | 1 | Software Reviews: A Formality Spectrum | S1:438-439 |
| 7 | 1 | Formal Technical Reviews<br><br>Metrics forprocessingproject | S1:441-444<br><br>S1:452-454 |
| 8 | 1 | Recapitulation and Discussion of important questions | |
| | **Total No of Hours Planned For Unit III** | | **8 Hours** |
| | | **UNIT-IV** | |
| 1 | 1 | Design Engineering<br>➢ Design within the Context of SoftwareEngineering<br>➢ Design Process and Design Quality | S1:225<br><br>S1:228-230 |

| | | | |
|---|---|---|---|
| 2 | 1 | Design Concepts<br>➢ Abstraction<br>➢ Architecture<br>➢ Patterns<br>➢ Modularity | S1:231-234<br>S5:138-146 |
| 3 | 1 | ➢ Information Hiding<br>➢ FunctionalIndependence<br>➢ Refinement<br>➢ Refactoring<br>➢ Design Classes | S1:235-239 |
| 4 | 1 | Architectural Design Elements<br>Creating an Architectural Design<br>➢ SoftwareArchitecture | S1:244-245<br><br>S1:252-256 |
| 5 | 1 | Data Design<br>➢ Data Design at the Architectural level<br>➢ Data Design at the Component level<br>➢ Refiningthe Architecture into Components<br>➢ Describing Instantiations of theSystem | <br>S2:289-291<br><br><br>S1:270-273 |
| 6 | 1 | MappingData Flow into a SoftwareArchitecture<br>➢ TransformFlow➢ TransactionFlow➢ Tra nsform Mapping<br>➢ Transaction Mapping<br>➢ Refiningthe Architectural Design | S2:307-315<br><br><br><br>S2:316-320 |
| 7 | 1 | ModelingComponentLevel Design<br>➢ Component<br>➢ DesigningClass-Based Components<br>➢ DesigningConventional Components | S2: 324-327<br>S2:330-353 |
| 8 | 1 | Recapitulation and Discussion of important questions | |
| | **Total No of Hours Planned For Unit 1V** | | **88Hours** |
| | | **UNIT-V** | |

| | | | |
|---|---|---|---|
| 1 | 1 | SoftwareTestingFundamentals<br>Strategic approach to software testing | S2:394-397<br>S1:466-472 |
| 2 | 1 | Testing strategies for conventional software<br>➢ Validation Testing | S1:473-481<br>S1:483-485 |
| 3 | 1 | System Testing<br><br>Black Box and White Box Testing<br>White BoxTesting<br>➢ Basis Path Testing | S1:486-487<br><br>S2:421-426<br>S1:500-506 |
| 4 | 1 | Control StructureTesting<br>BlackBoxTesting<br>➢ Graph Based Testing Methods<br>➢ Equivalencepartitioning<br>➢ BoundaryValueAnalysis<br>➢ Orthogonal Arraytesting | S1:507-509<br><br>S1:509-512<br><br>S5:463-466<br>S1:512-516 |
| 5 | 1 | Recapitulation and Discussion of important questions | |
| 6 | 1 | **DiscussionofpreviousESEQuestionpapers** | |
| 7 | 1 | **DiscussionofpreviousESEQuestionpapers** | |
| 8 | 1 | **DiscussionofpreviousESEQuestionpapers** | |
| | **Total No of Hours Planned For Unit V** | | **8 Hours** |
| | **Total Planned Hours** | | **40 Hours** |

**SUGGESTEDREADINGS**

**S1:** Roger S. Pressman, 2015, SoftwareEngineering– A Practitioner's Approach, 8th Edition,McGrawHillInternational Edition, New Delhi.

**S2:** R.S. Pressman, (2009).SoftwareEngineering: A Practitioner's Approach (7thed.). McGraw-Hill.

**S3:**P.Jalote (2008). AnIntegratedApproach to SoftwareEngineering(2nded.). New Age International Publishers.

**S4:** K.K. AggarwalandY.Singh (2008).Software Engineering(2nd ed.). New Age International Publishers.

**S5:**Sommerville(2006). SoftwareEngineering(8th ed.). Addison Wesley.

**S6:**D.Bell (2005). SoftwareEngineeringforStudents (4th ed.)Addison-Wesley.

**S7:**R.Mall (2004).FundamentalsofSoftwareEngineering(2nd ed.). Prentice-HallofIndia.

**WEBSITES**

**W1**: https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc

**W2**: en.wikipedia.org/wiki/

**W3**: http://www.slideshare.net/neelamani/software-engineering-note

**W4**: www.iso.org/iso/iso_9000

**W5**:https://www.technicalcommunicationcenter.com/2010/07/19/10-characteristics-of-high-quality-srs-software-requirements-specifications/

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE : 17CTU402** **UNIT I** **SEMESTER : IV**
**BATCH (2017-2020)**

**UNIT-I**

**Introduction:** The Evolving Role of Software, Software Characteristics, Changing Nature of Software, Software Engineering as a Layered Technology, Software Process Framework, Framework and Umbrella Activities, Process Models, Capability Maturity Model Integration (CMMI).

## Introduction to Software Engineering:

### What is software engineering?

Software has become critical to advancement in almost all areas of human Endeavour. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products. Software engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

### Definition

At the first conference on software engineering in 1968, Fritz Bauer [FRIT68] defined software engineering as *"The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines"*. Stephen Schacht [SCHA90] defined the same as *"A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements"*. Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

### The Evolving Role of Software

Software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local

hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer— producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information.

Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

**Software**

**Computer software**, or just **software**, is a collection of computer programs and related data that provide the instructions for telling a computer what to do and how to do it. In other words, software is a conceptual entity which is a set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system. We can also say software refers to one or more computer programs and data held in the storage of the computer for some purposes.

In other words software is a set of **programs, procedures, algorithms** and its **documentation**. Program software performs the function of the program it implements,

either by directly providing instructions to the computer hardware or by serving as input to another piece of software.

The term was coined to contrast to the old term hardware (meaning physical devices). In contrast to hardware, software is intangible, meaning it "cannot be touched". Software is also sometimes used in a more narrow sense, meaning application software only. Sometimes the term includes data that has not traditionally been associated with computers, such as film, tapes, and records.

**Software Characteristics**

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

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different . 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."**

Fig 1.1 depicts failure rate as a function of time for hardware.

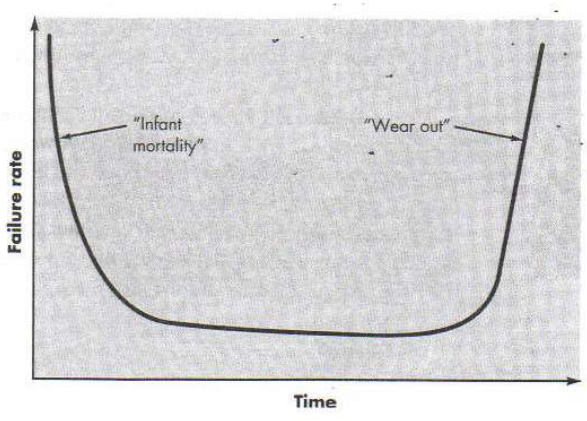


**Fig 1.1 Failure curve for hardware**

The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

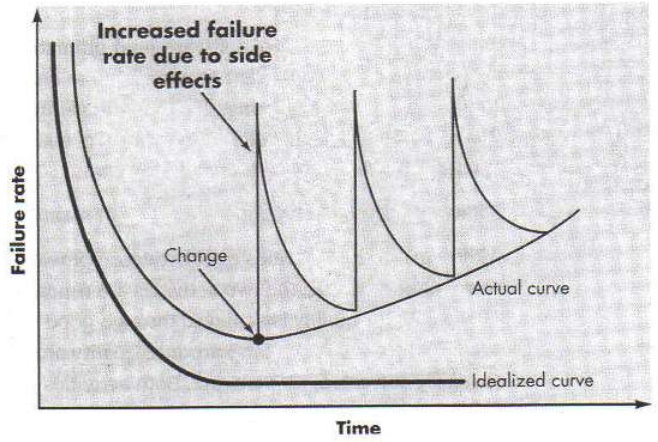



**Fig 1.2 Failure curves for software**

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Fig 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out.

This seeming contradiction can best be explained by considering the "actual curve" shown in Fig 1.2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Fig 1.2. 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.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

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

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an *IC* or a *chip*) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure

and processing detail required to build the interface are contained with a library of reusable components for interface construction.

**Software Myths**

Belief about the software and the process used to build it- can be traced To the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners

**Management myths**

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

***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? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality?

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

***Reality:*** Software development is not a mechanistic process like manufacturing.

In the words of Brooks [BRO75]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People

can be added but only in a planned and well-coordinated manner

***Myth:*** If I decide to outsource3 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 out sources software projects

**Customer myths**

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

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

***Reality:*** Although a comprehensive and stable statement of requirements is not always possible an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer

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

***Reality:*** It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early cost impact is relatively small. However, as time passes, cost impact grows rapidly-resources have been committed, a design framework has been established and a change can cause upheaval that requires additional resources and major design modification

**Practitioner's myths**

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form.

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

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

***Myth:*** 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 *formal technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

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

***Reality:*** A working program is only one part of a *software configuration* that includes many elements. Documentation provides 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 quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

## A Generic View of process

### Software Engineering as a Layered Technology

Any engineering approach much rests on organizational approach to quality, e.g. total quality management and such emphasize continuous process improvement (that is increasingly more effective approaches to software engineering). The bedrock that supports a software engineering is a ***quality focus***.

The foundation for software engineering is the ***process*** layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of *key process areas* (KPAs) that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.


**Fig 1.3 Software Engineering Layers**

Software engineering ***methods*** provide the technical how-to's for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

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

**Process Framework**

Identifies a small number of framework activities that are applicable to all software projects. In addition the framework encompasses umbrella activities that are applicable across the software process.

**Generic Process Framework Activities**

Each framework activity is populated by a set of software engineering actions. An action, e.g. design, is a collection of related tasks that produce a major software engineering work product.

**Communication** – lots of communication and collaboration with customer and other stakeholders.. Encompasses requirements gathering.

**Planning** – establishes plan for software engineering work that follows. Describes technical tasks, likely risks, required resources, works products and a work schedule

**Modeling** – encompasses creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements.

**Modeling Activity – composed of two software engineering actions**

• **analysis** – composed of work tasks (e.g. requirement gathering, elaboration, specification and validation) that lead to creation of analysis model and/or requirements specification.

• **design** – encompasses work tasks such as data design, architectural design, interface design and component level design leads to creation of design model and/or a design specification.

**Construction** – code generation and testing.

**Deployment** – software, partial or complete, is delivered to the customer who evaluates it and provides feedback.

Different projects demand different task sets. Software team chooses task set based on problem and project characteristics.

**Process Models**

**Prescriptive Model**

Every software engineering organization should describe a unique set of framework activities for the software process it adopts. It should populate each framework activity with a set of software engineering actions, and define each action in terms of a task set that identifies the work to be accomplished to meet the development goals.

It should then adapt the resultant process model to accommodate the specific nature of each project, the people who will do the work, and the environment in which the work will be conducted. Regardless of the process model that is selected, software engineers have traditionally chosen a generic process framework that encompasses the following framework activities: communication, planning, modeling, construction, and deployment.

We call them "prescriptive" because they prescribe a set of process elements framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a workflow- that is, the manner in which the process elements are inter-related to one another.

All software process models can accommodate the generic framework activities that have been described, but each applies a different emphasis to these activities and defines a workflow that invokes each framework activity in a different manner.

**Waterfall Model**

There are times when the requirements of a problem are reasonably 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. It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable

The waterfall model, sometimes called the *classic life cycle model,* suggests a systematic, sequential approach to software development that begins with customer

specification of requirements and progresses through planning, modeling, construction, and deployment.

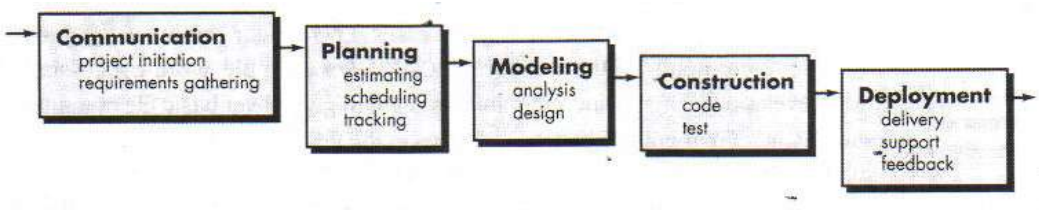



**Fig 1.4 Waterfall Model**

The waterfall model is the oldest paradigm for software engineering. However, over the past two decades, criticism of this process model has caused even ardent supporters to question its efficacy. 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. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

**2.** It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

**3.** The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

In an interesting analysis of actual projects found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never –ending stream of changes. The waterfall model is often inappropriate for such work. However, 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.

**Incremental process Models**.

There are many situations in which initial software requirements are reasonably well-defined, but the overall scope of the development effort precludes a purely linear process. In addition, 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, a process model that is designed to produce the software in increments is chosen.

1. **The Incremental Model**

The *incremental model* combines elements of the waterfall model applied in an iterative fashion. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable "increment" of the software.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm

When an incremental model is used, the first increment is often a *core product.*
That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). 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

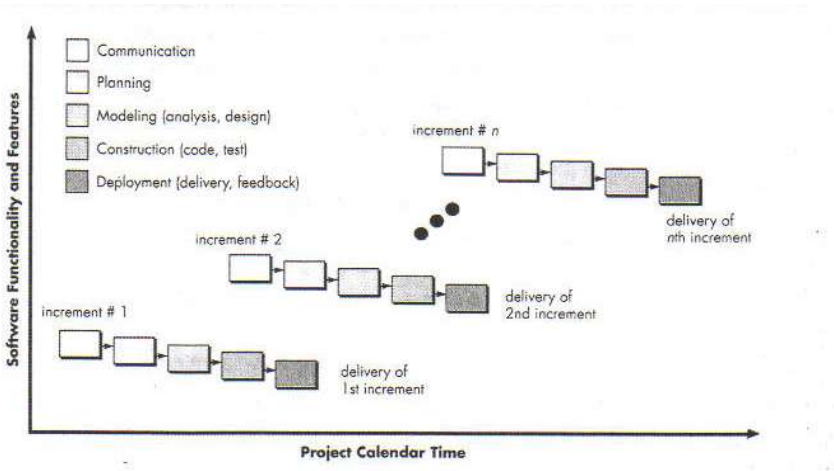



**Fig:1.5 The Incremental Model**

The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

**Increment 1:** Analysis-->Design-->Code-->Test (Delivery of 1st Increments. Normally '"Core Product"')

**Increment 2:** Analysis-->Design-->Code-->Test (Delivery of 2nd Increments)

**Increment n:** Analysis-->Design-->Code-->Test (Delivery of nth Increments)

**Advantages**

- It is useful when staffing is unavailable for the complete implementation.
- Can be implemented with fewer staff people.
- If the core product is well received then the additional staff can be added.
- Customers can be involved at an early stage.
- Each iteration delivers a functionally operational product and thus customers can get to see the working version of the product at each stage.

## 2. The RAD Model

*Rapid application development* (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaptation of the waterfall model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days)

Like other process models, the RAD approach maps into the generic framework activities.

- Communication works to understand the business problem and the information characteristics that the software must accommodate.
- Planning is essential because multiple software teams work in parallel on different system functions.
- Modeling encompasses three major phases- business modeling, data modeling and process modeling and establishes design representations that serve as the basis for RAD's construction activity.
- Construction emphasizes the use of preexisting software components and the application of automatic code generation.
- Finally, deployment establishes a basis for subsequent iterations, if required.

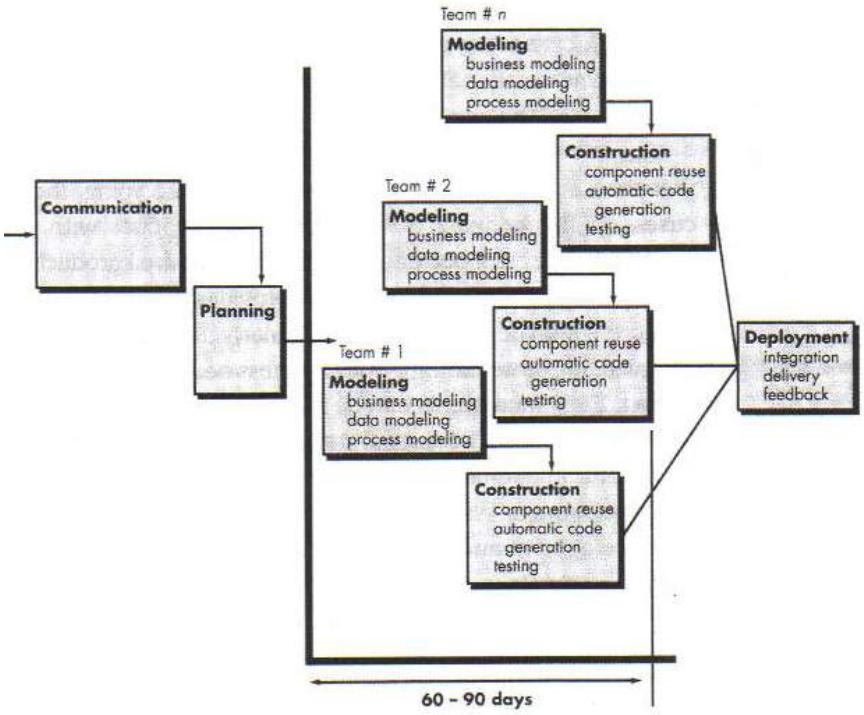
The RAD process model is illustrated in Fig 1.6. Obviously, the time constraints imposed on a RAD project demand "scalable scope" . If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

**Drawbacks**

• For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

• If developers and customers are not committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame, RAD projects will fail.

**Fig:1.6 The RAD Model**




•. If a system cannot be properly modularized, building the components necessary for RAD will be problematic

- If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.

• RAD may not be appropriate when technical risks are high.

**Evolutionary Process Model**

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In

these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

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

1. **Prototyping**

**Software prototyping**, refers to the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

A prototype typically simulates only a few aspects of the final solution, and may be completely different from the final product.

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system,

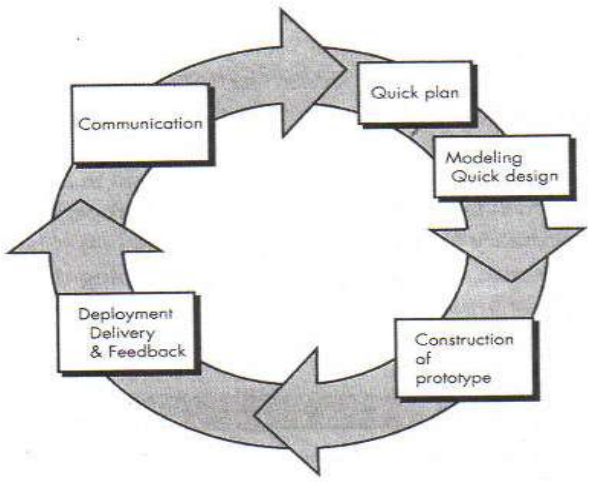
or the form that human/machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a standalone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. Regardless of the manner in which it is applied, the prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly and modeling (“in the form of ”quick design”)

occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats).

The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer. Feedback is used to refine requirements for the software.




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

**Advantages**

1.The software designer and implementer can obtain feedback from the users early in the project

2.The client and the contractor can compare if the software made matches the software specification, according to which the software program is built.

3.It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met.

***Disadvantages***

1.Often clients expect that a few minor changes to the prototype will more than suffice their needs. They fail to realise that no consideration was given to the overall quality of the software in the rush to develop the prototype.

2.The developers may lose focus on the real purpose of the prototype and compromise the quality of the product. For example, they may employ some of the inefficient algorithms or inappropriate programming languages used in developing the prototype. This mainly due to laziness and an over reliance on familiarity with seemingly easier methods.

3. A prototype will hardly be acceptable in court in the event that the client does not agree that the developer has discharged his/her obligations. For this reason using the prototype as the software specification is normally reserved for software development within an organisation.

2. **The Spiral Model**

The Spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the Waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.

Using the Spiral Model the software is developed in a series of evolutionary releases**.** During early iterations, the release might be a prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A Spiral Model is divided into a number of framework activities defined by the software engineering team. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center

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

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery.

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

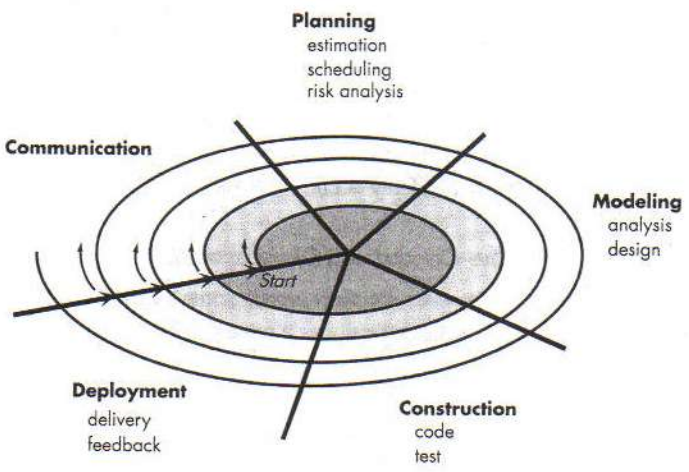



**Fig:1.8 The Spiral Model**

**Advantages of the Spiral Model**

- Realistic approach to the development because the software evolves as the process progresses. In addition, the developer and the client better understand and react to risks at each evolutionary level.
- The model uses prototyping as a risk reduction mechanism and allows for the development of prototypes at any stage of the evolutionary development.
- It maintains a systematic stepwise approach, like the classic waterfall model, and also incorporates into it an iterative framework that more reflect the real world.

**Disadvantages of the Spiral Model**

- One should possess considerable risk-assessment expertise
- It has not been employed as much proven models (e.g. the Waterfall Model) and hence may prove difficult to 'sell' to the client.

**3. The Concurrent Development Model**

The concurrent development model, sometimes called *concurrent* engineering can be represented schematically as a series of framework activities, software engineering actions and tasks, and their associated states. For example, the modeling activity defined for the spiral model is accomplished by invoking the following actions: prototyping and/or analysis modeling and specification and design.

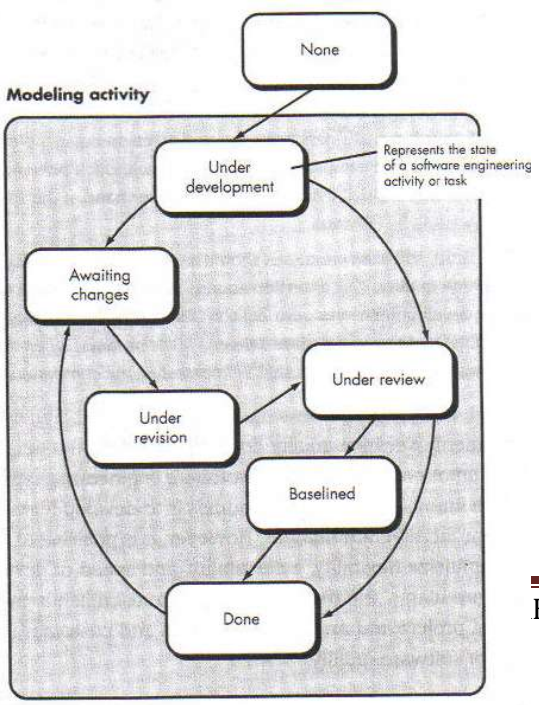


**Fig:1.9 One element of the Concurrent process Model**

The activity—analysis—may be in any one of the states noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the *customer communication* activity (not shown in the figure) has completed its first iteration and exists in the awaiting changes state. The *analysis* activity (which existed in the none state while initial customer communication was completed) now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the *analysis* activity moves from the under development state into the awaiting changes state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. For example, during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event *analysis model correction* which will trigger the *analysis* activity from the **done** state into the **awaiting changes** state.

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

**4. A Final Comment on the Evolutionary Processes**

The first concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on the linear layouts of activities, so they do not fit completely

Second, Evolutionary processes do not establish the maximum speed of the evolution. On the other hand, if the speed is too slow then productivity could be affected.

Third, software processes should be focused on flexibility and extensibility rather than on high quality

The intent of Evolutionary process model is to develop high quality software in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility and speed of development. The challenge for software teams and project managers is to establish a proper balance between these critical project and product parameters and customer satisfaction

**Specialized Process Models**

Specialized models tend to be applied when a narrowly defined software engineering approach is chosen

**1. Component Based Development**

Commercial off the shelf (COTS) software components, developed by vendors, who offer them as products, can be used when software is to be built. These components provide targeted functionality with well defined interfaces that enable the component to be integrated into the software.

The component-based development (CBD) model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components (called *classes*).

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object oriented classes

The component based component model incorporates the following steps

- Available component based products are researched and evaluated for the application domain
- Component integration issues are considered
- Software architecture is designed to accommodate the components

- Components are integrated into the architecture
- Comprehensive testing is conducted to ensure proper functionality

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Based on studies of reusability, QSM Associates, Inc., reports component assembly leads to a 70 percent reduction in development cycle time; an 84 percent reduction in project cost.

**2. The Formal Methods Model**

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms.

Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily, not through ad hoc review but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might go undetected.

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

**3. Aspect oriented Software Development**

Regardless of the software process that is chosen, the builders of the complex software invariably implement a set of lacalized features, functions and information content. These localized software characteristics are modeled as components and then constructed within the context of a system architecture

When concerns cut across multiple system functions, features and information, they are often referred to as crosscutting concerns.

Aspect oriented software development(AOSD) often referred to as aspect oriented programming(AOP) is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing and constructing aspects

Aspect oriented component engineering(AOCE) uses a concept of horizontal slices through vertically decomposed software components called aspects to characterize cross cutting functional and non functional properties of components

Important Questions

1. What is Software Engineering?
2. Describe the Layered Approach of Software Engineering.
3. Explain about the Process Framework activities.
4. Illustrate about the Process model and its type.
5. Explain about the RAD model.
6. Define CMMI.
7. Define the nature of Software Engineering.

| | **Software Enginering (17CTU402)** | | **UNIT I** | | |
|---|---|---|---|---|---|
| **S.No** | **Question** | **Option A** | **Option B** | **Option C** | **Option D** |
| 1 | Software takes on a _____ role. | single | dual | triple | tetra |
| 2 | Software is a _____. | virtual | system | modifier | framework |
| 3 | Instructions that when executed provide desired function and performance is called | software | hardware | firmware | humanware |
| 4 | High quality of software is achieved through _____. | testing | good design | construction | manufacture |
| 5 | Software doesn't _____. | tearout | wearout | degrade | deteriorate |
| 6 | Software is not susceptible to _____. | hardware | defects | environmental melodies | deterioration |
| 7 | Software will undergo _____. | database | testing | enhancement | manufacture |
| 8 | _____ refers to the meaning and form of incoming and outgoing | content | software | hardware | data |
| 9 | _____ refers to the predictability of the order and timing of | system software | network software | information determinacy | database |
| 10 | _____ is not a system software. | MS Office | compiler | editor | file management utility |

| | | | | | |
|---|---|---|---|---|---|
| 11 | Collection of programs written to service other programs are called _____. | system software | business software | embedded software | d. pc software |
| 12 | Which one is not coming under software myths | Management myths | customer myths | product myths | practitioners myths |
| 13 | _____ is a PC Software. | MS word | LISP | CAD | C |
| 14 | Software that monitors, analyses, controls real world events is called _____. | Business software | real time software | web based software | d. embedded software |
| 15 | The bedrock that supports software engineering is a_____ | tools | methods | process models | a quality focus |
| 16 | A complete software process by identifying a small number of _____ | framework activities | umbrella activities | process framework | software process |
| 17 | The process framework encompassess a set of _____ | framework activities | umbrella activities | process framework | software process |
| 18 | software engineering action . | design | chronic | decision | crisis |
| 19 | Which one is effect the outcome of the project? | Risk management | Measurement | technical reviews | Reusability |
| 20 | Continuing indefinitely is called _____. | crisis | decision | affliction | chronic |
| 21 | Component based development uses | functions | subroutines | procedures | objects |
| 22 | UML stands for _____. | Universal Modelling Language | User Modified Language | Unified Modelling Language | User Model Language |
| 23 | A model which uses formal mathematical specification is called _____. | 4 GT model | Unified method model | formal methods model | component based development |
| 24 | A variation of formal methods model is called _____. | component based development | 4 GT model | unified method model | cleanroom software engineering |

| | | | | | |
|---|---|---|---|---|---|
| 25 | The development of formal methods is _____. | less time consuming | quite time consuming | does not consume time | very less time consuming |
| 26 | The first step to develop software is _____. | analysis | design | requirements gathering | coding |
| 27 | The waterfall model sometimes called as | classic model | classic life cycle model | life cycle model | cycle model |
| 28 | Software engineering activities include | decision | affliction | hardware | maintenance |
| 29 | all process model prescribes a | circular | elliptical | spiral | workflow |
| 30 | Component based development incorporates the characteristics of the _____ model | circular | elliptical | spiral | hierarchical |
| 31 | Prototype is a _____. | software | hardware | computer | model |
| 32 | For small applications it is possible to move from requirement gathering step | analysis | implementation | design | modeling |
| 33 | Software project management begins with a | project planning | software scope | software estimation | decomposition |
| 34 | Breaking up of a complex problem into small steps is called _____. | project planning | software scope | software estimation | decomposition |
| 35 | The ease with which software can be transferred from one computer to another. This | portability | reliability | efficiency | accuracy |
| 36 | The ability of a program to perform a required function under stated | portability | reliability | efficiency | accuracy |
| 37 | The event to which software performs its intended function. This quality attribute is called | portability | reliability | efficiency | accuracy |
| 38 | A qualitative assessments of freedom from errors. This quality attribute is called | portability | reliability | efficiency | accuracy |
| 39 | The extent to which software can continue to operate correctly. This quality attribute is called | robustness | correctness | efficiency | reliability |

| | | | | | |
|---|---|---|---|---|---|
| 40 | The extent to which the software is free from design and coding defects is fault free. This quality | robustness | correctness | efficiency | reliability |
| 41 | System shall reside in 50KB of memory is an example of | quantified requirement | qualified requirement | functional requirement | performance requirement |
| 42 | Accuracy shall be sufficient to support mission is an example of | quantified requirement | qualified requirement | functional requirement | performance requirement |
| 43 | System shall make efficient use of memory is an example of | quantified requirement | qualified requirement | functional requirement | performance requirement |
| 44 | A software product often has | Multiple users | developers | users | developers and maintainers. |
| 45 | Multiprogramming and time sharing software techniques were developed during the | first generation computing | second generation computing | third generation computing | fifth generation computing |
| 46 | According to Boehm software engineering involves the practical application of scientific | Planning of computer programs | analysis of computer programs | design & construction of computer programs | design & maintainer of computer programs. |
| 47 | IEEE define software engineering as the systematic approach to the development operation | implementation of the software. | retirement of the slw | construction of the software | maintenance |
| 48 | Good, oral, written and---- skills are crucial for the software engineer | interpersonal communication | communication | managerial skills | MIS |
| 49 | Software is ------- | changeable | modified | updateable | intangible |
| 50 | In software engineering the unit of decomposition are called | units | modules | relationships | components |
| 51 | Control interfaces are established by calling ------ among modules | global data items | functions | relationships | local data items |
| 52 | programmers who intentionally write convoluted programs that have obscure side effects are known as ---- | intruders | analyzers | testers | hackers |

| | | | | | |
|---|---|---|---|---|---|
| 53 | ------ is used to denote an individual who is concerned with the details of implementing packaging and modifying algorithms and | system analysis | programmer | software engineer | customer. |
| 54 | ------- are additionally concerned with issues of analysis, design, | Developers | analyst | customers | software engineers. |
| 55 | On large projects ----- are essential | analysis & design | implementation & testing | modification | standard practices& formal |
| 56 | The term " computer software" is often take | Project | programs | collection of programs | source code. |
| 57 | Software products include | System level software | application programs | System level software & application | OS |
| 58 | Documentation explains the ------- | contact of the project | modules of the project | characteristic of an | Software usage |
| 59 | ------ is a primary concern of software engineers | Software design | software maintenance | software product | software quality. |
| 60 | The quality attributes for very software product includes | design | clarity | accuracy | visibility |

| | | |
|---|---|---|
| | | |
| | | **Answer** |
| | | dual |
| | | modifier |
| | | software |
| | | good design |
| | | wearout |
| | | environmental melodies |
| | | enhancement |
| | | content |
| | | information determin |
| | | MS Office |

| | | |
|---|---|---|
| | | system software |
| | | product myths |
| | | MS word |
| | | real time software |
| | | a quality focus |
| | | framework activities |
| | | umbrella activities |
| | | design |
| | | Risk management |
| | | chronic |
| | | objects |
| | | Unified Modelling |
| | | formal methods model |
| | | cleanroom software |

| | | |
|---|---|---|
| | | quite time consumin |
| | | requirements gathering |
| | | classic life cycle model |
| | | maintenance |
| | | workflow |
| | | spiral |
| | | model |
| | | implementation |
| | | project planning |
| | | decomposition |
| | | portability |
| | | reliability |
| | | efficiency |
| | | accuracy |
| | | robustness |

| | | |
|---|---|---|
| | | correctness |
| | | quantified requirem |
| | | qualified requirement |
| | | qualified requirem |
| | | developers and maintaine |
| | | third generation computin |
| | | design & construction of |
| | | retirement of the |
| | | interpersonal communication |
| | | intangible |
| | | modules |
| | | relationships |
| | | hackers |

| | | |
|---|---|---|
| | | programmer |
| | | software engineers. |
| | | standard practices & formal |
| | | source code. |
| | | System level software |
| | | characteristic of an |
| | | software quality. |
| | | clarity |

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE : 17CTU402** **UNIT II** **SEMESTER : IV**
**BATCH (2017-2020)**

**UNIT-II**

**Requirement Analysis;** Initiating Requirement EngineeringProcess- Requirement Analysis and Modeling Techniques- FlowOriented Modeling- Need for SRS- Characteristics and Components of SRS- Software Project Management: Estimation in Project Planning Process, Project Scheduling.

## Requirements Engineering

**Requirements analysis** in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users. It is an early stage in the more general activity of requirements engineering which encompasses all activities concerned with eliciting, analyzing, documenting, validating and managing software or system requirements.

Requirements analysis is critical to the success of a systems or software project. The Requirements should be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

## Requirements Engineering Tasks

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system

The requirement engineering process is accomplished through the execution of seven distinct functions. They are

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification

- Validation
- Management

**Inception**

How does a software project get started? Is there a single event that becomes the catalyst for a new computer based system or product, or does the need evolve over time?

Stakeholders from the business community define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope.

At project inception, software engineer's ask a set of context free questions discussed. The intent is to establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the customer and developer

**Elicitation**

It certainly seems simple enough-ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day to day basis

**i) Problem of scope**

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

**ii) Problem of understanding**

The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer or specify requirements that are ambiguous

**iii) Problem of volatility**

The requirements change over time

**Elaboration**

The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This requirement engineering activity focuses on developing a refined technical model of software functions, features, and constraints

Elaboration is an analysis modeling action that is composed of a number of modeling and refinement tasks. Elaboration is driven by the creation and refinement of user scenarios that describe how the end-user interacts with the system. Each user scenario is parsed to extract analysis classes-business domain entities that are visible to the end user. The attributes of each analysis classes are defined and the services that are required by each class are identified.

The end result of elaboration is an analysis model that defines the informational, functional, and behavioral domain of the problem

**Negotiation**

It is also relatively common for different customers or users to propose conflicting requirements, arguing that their version is essential for our special needs.

The requirement engineer must reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Risk associated with each requirement are identified and analyzed. Using an iterative approach, requirements are eliminated, combined, and modified so that each party achieves some measure of satisfaction

**Specification**

A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype or any combination of these.

However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.

The specification is the final work product produced by requirement engineer. It serve as the foundation for subsequent software engineering activities. It describes the function and performance of a computer-based system and the constraints that will govern its development

**Validation**

The work product produced as a consequence of requirements engineering are assessed for quality during validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; inconsistencies, omissions and errors have been detected and corrected

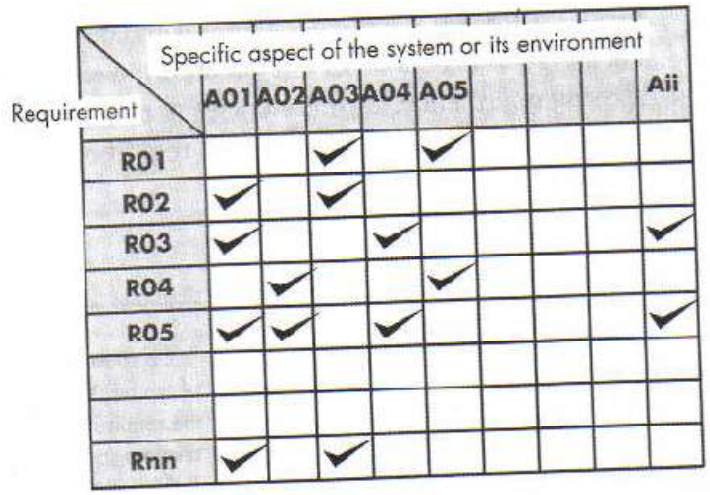
The primary requirement validation mechanism is the formal technical review. The review team that validates requirements includes software engineers, customers, users and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements or unrealistic requirements

**Requirements Management**

Requirements management is the set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management techniques

Requirements management begins with identification. Each requirement is assigned a unique identifier. Once requirements have been identified, traceability tables

are developed. Each traceability table relates requirements to one or more aspects of the system or its environment




**Traceability table**

**i) Features traceability table**

It shows how requirements relate to important customer observable system

**ii) Source traceability table**

Identifies the source of each requirement

**iii) Dependency traceability table**

Indicates how requirements are related to one another

**iv) Subsystem traceability table**

Categorizes requirements by the subsystems that they govern

**v) Interface traceability table**

Shows how requirements relate to both internal and external system interfaces

**Initiating the Requirement Engineering Process**

The steps required to initiate requirements engineering-to get the project started in a way that will keep it moving forward toward a successful solution

**i) Identifying the stakeholders**

Sommerville and Sawyer define a stakeholder as “anyone who benefits in a direct or indirect way from the system which is being developed”. We have already identified

the usual suspects: business operations manager, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others.

Every stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail

**ii) Recognizing multiple viewpoints**

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End-users may want features that are familiar to them and that are easy to learn and use.

Each of these constituencies will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. The job of the requirements engineer is to categorize all stakeholder information in a way that will allow decision makers to choose an internally consistent set of requirements for the system

**iii) Working toward collaboration**

The job of the requirements engineer is to identify areas of commonality and areas of conflict or inconsistency

Collaboration does not necessarily mean that requirements are defines by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong “project champion” may take the final decision about which requirements make the cut

**iv) Asking the first questions**

The first set of context-free questions focuses on the customer and other stakeholders, overall goals, and benefits.

For example, the requirements engineer might ask

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development

The next set of questions enables the software team to gain a better understanding of the problem

- How would you characterize "good" output that would be generated by a successful solution?
- What problems will this solution address?
- Can you show me the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself

- Are you the right person to answer these questions?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

**Eliciting Requirements**

The Q & A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, elaboration, negotiation and specification.

**i) Collaborative requirements gathering**

In order to encourage a collaborative, team-oriented approach to requirements gathering, a team of stakeholders and developers work together to identify the problem, propose elements of the solution

Many different approaches to collaborative requirements gathering have been proposed

- Meetings are conducted and attended by both customers and software engineers
- Rules for preparation and participation are established
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas
- A "facilitator" controls the meeting
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conductive to the accomplishment of the goal

To better understand the flow of events as they occur, we present a brief scenario that outlines the sequence of events that lead up to the requirements gathering meeting, occur during the meeting, and follow the meeting

During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings the stakeholders write a one-or two-page "product request". Members of the software team and other stakeholder organizations are invited to attend. The product request is distributed to all attendees before the meeting date

**ii) Quality function deployment**

Quality function deployment is a technique that translates the needs of the customer into technical requirements for software. It concentrates on maximizing customer satisfaction from the software engineering process. Quality function deployment identifies three types of requirements

**a) Normal requirements**

These requirements reflect objectives and goals stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied.

**Example:** Requested type of graphical displays, specific system functions and defined levels of performance

**b) Expected requirements**

These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them.

**Example**: overall operational correctness and reliability

**c) Exciting Requirements**

These requirements reflect features that go beyond the customer's expectations and prove to be very satisfying when present

**Example:** Word processing software is requested with standard features

Quality function deployment uses customer reviews and observation, surveys and examination of historical data as raw data for the requirements gathering activity. These data are then translated into a table of requirements- called the customer voice table-that is reviewed with the customer

A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements

**iii) User Scenarios**

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until the software team understands how these functions and features will be used by different classes of end-users. To accomplish this , developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use-cases, provide a description of how the system will be used.

**iv) Elicitation Work Products**

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems,the work products include:

- A statement of need and feasibility

- A bounded statement of scope for the system or product
- A list of customers, users and other stakeholders who participated in requirements elicitation
- A description of the system's technical environment
- A list of requirements and the domain constraints that apply to each
- A set of usage scenario that provide insight into the use of the system or product under different operating conditions
- Any prototypes developed to better define requirements

**Building the Analysis Model**

**Requirement Analysis**

Requirement Analysis results in the specification of software's operational characteristics indicates software interface with other system elements and establishes constraints that software must meet

Requirement analysis allow the software engineer to elaborate on basic requirements established during earlier requirement engineering tasks and build models that depict user scenario, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

Requirement analysis provides the software designer with a representation of information, function and behavior that can be translated to architectural, interface and component-level designs

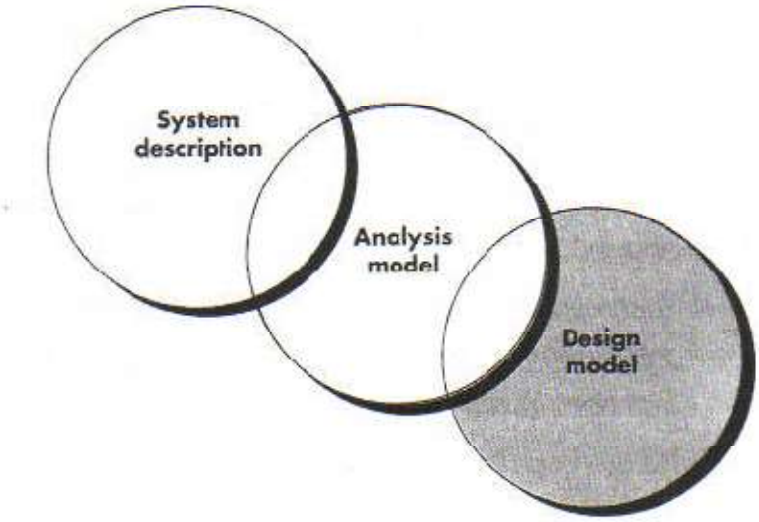
Finally, the analysis model and the requirement specification provide the developer and customer with the means to assess quality once software is built. Throughout analysis modeling, the software engineer's primary focus is on what and not how

**1. Overall Objectives and Philosophy**

The analysis model must have three primary objectives

- To describe what the customer requires
- To establish a basis for the creation of software design
- To define a set of requirements that can be validated once the software is built

The analysis model bridges the gap between a system level description that describes overall system functionality as it is achieved by applying software, hardware, data, human and other system elements and a software design that describes the software's application architecture, user interface and component level structure



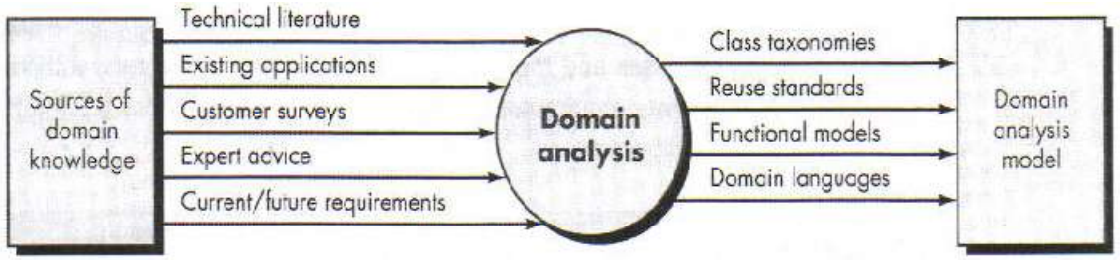

**2. Analysis rules of Thumb**

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system
- Delay consideration of infrastructure and non functional models until design
- Minimize coupling throughout the system
- Be certain that the analysis model provides value to all stakeholders
- Keep the model as simple as it can be

**3. Domain Analysis**

The analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows a

software engineer or analyst to recognize and reuse them, the creation of the analysis model is expedited.



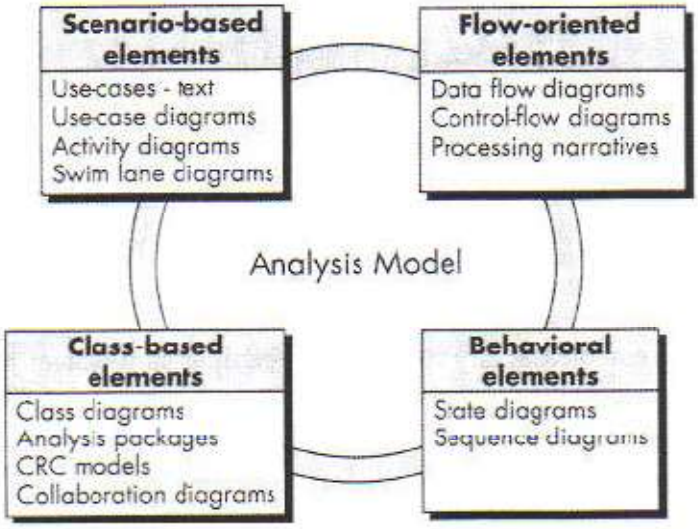

Input and output of Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within the application domain.

**Analysis modeling approaches**

One view of analysis modeling, called structured analysis, considers Data and the processes that transform the dada as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as a data flow through the system.

A second approach to analysis modeling, called objects oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.




Analysis Modeling Approaches

**Data Modeling Concepts**

Analysis modeling often begins with data modeling. The software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.

**1. Data object**

A *data object* is a representation of almost any composite information that must be understood by software. By *composite information,* we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example,

a person or a car (Figure 12.2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

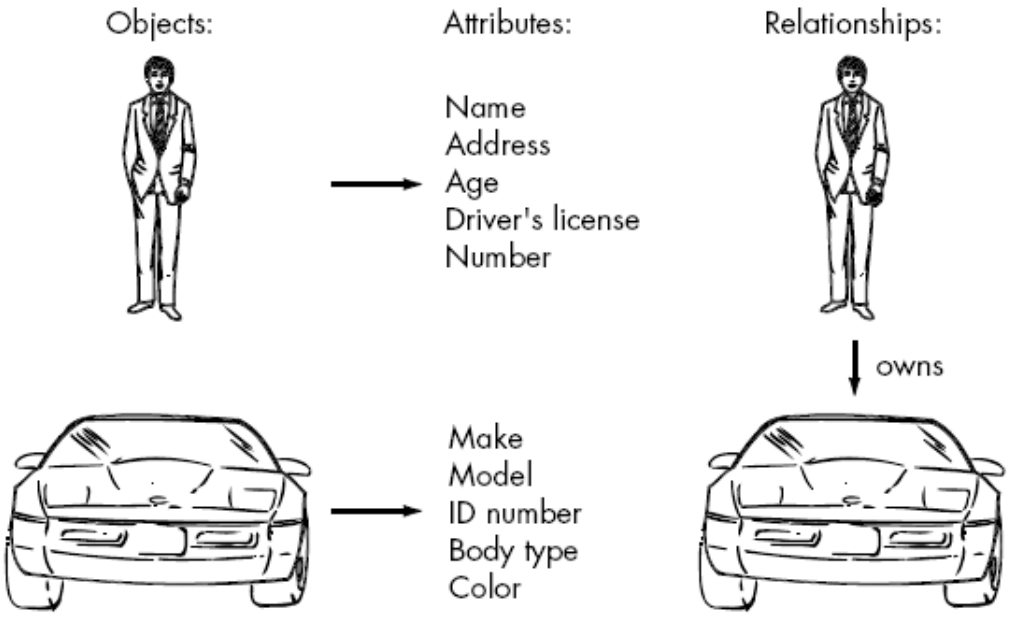
**2. Data Attributes**

Attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

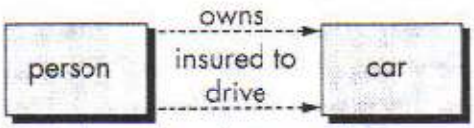
(3) make reference to another instance in another table.




**3. Relationships**

Data objects are connected to one another in different ways. Consider two data objects, person and car. These objects can be represented using the simple notation illustrated in below Figure. A connection is established between person and car because the two objects are related.


(a) A basic connection between data objects




(b) Relationships between data objects

**4. Cardinality and Modality**

The elements of data modeling—data objects, attributes, and relationships—provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: **object X** *relates* to **object Y** does not provide enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called *cardinality.*

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

Cardinality defines "the maximum number of objects that can participate in a relationship"

**Modality**

The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

## Flow-Oriented Modeling

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows and the transformations are represented by circles (also called bubbles). The DFD is presented in hierarchical fashion. That is, the first data flow model sometimes called a level 0 DFD or context diagram represent the system as a whole.

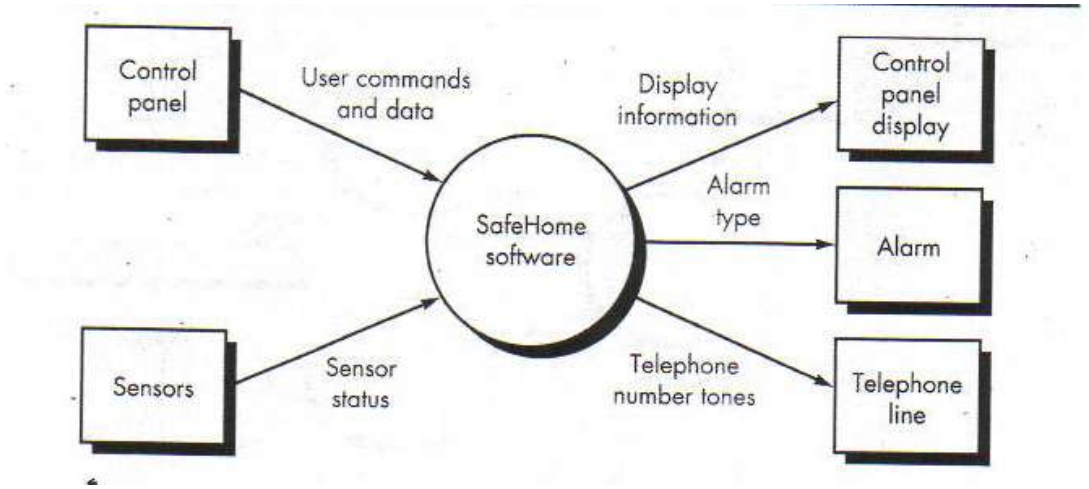
### 1. Creating a data flow model

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

**Guidelines**

1. The level 0 data flow diagram should depict the software/system as a single bubble
2. Primary input and output should be carefully noted
3. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level
4. All arrows and bubbles should be labeled with meaningful names
5. Information flow continuity must be maintained from level to level
6. One bubble at a time should be refined.

**Context level DFD for the safe home security function**



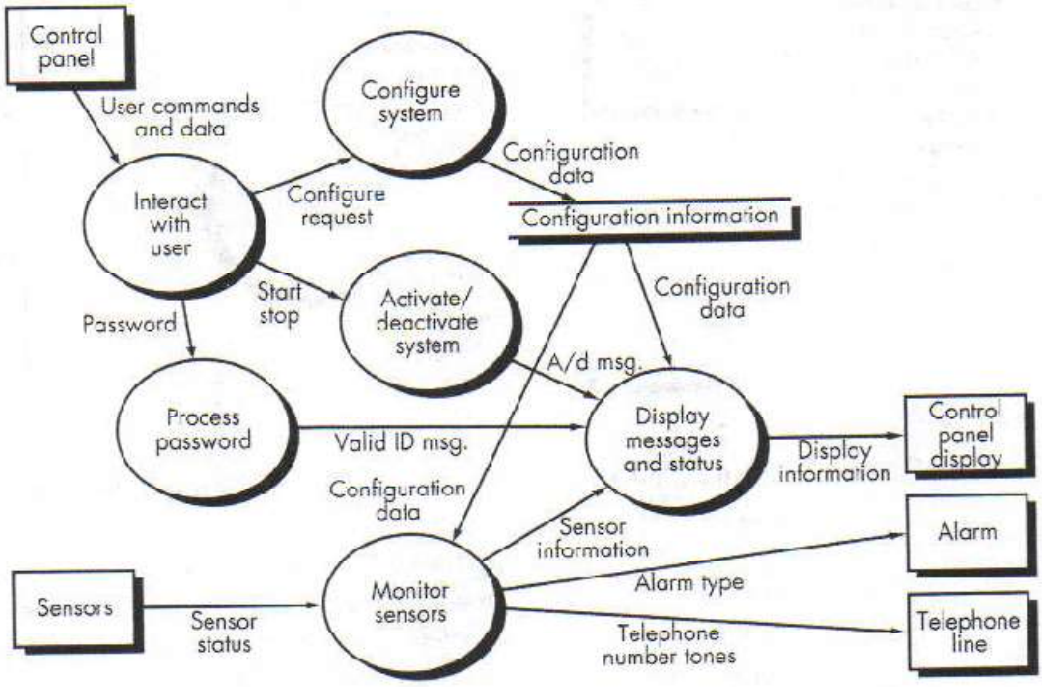

The safe home security function enables the homeowner to configure the security system. When it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through the internet, a PC, or a control panel

During installation, the safe home PC is used to program and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

When a sensor event is recognized, the software involves an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until a telephone connection is obtained

The level 0 DFD is now expanded into a level 1 data flow model

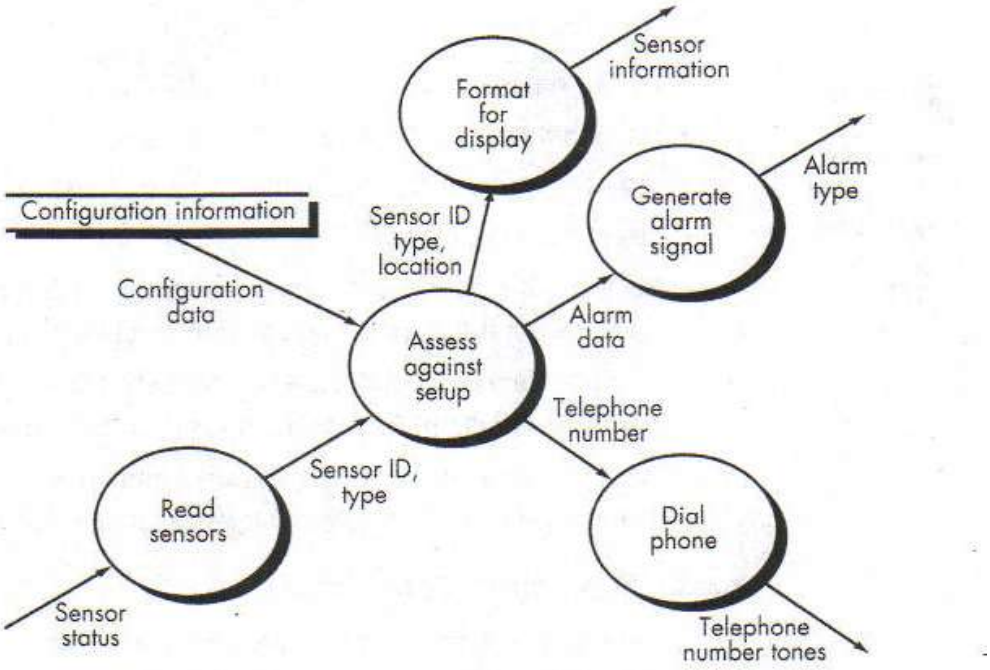
**Level 1 DFD for the safe home security function**




The homeowner receives security information via a control panel, the PC, or a browser, collectively called an interface. The interface displays prompting messages and system status information on the control panel, the PC, or the browser window

The process represented at DFD level 1 can be further refined into lower levels. For example, the process monitor sensors can be refined into a level 2 DFD.

**Level 2 DFD that refines the monitor sensors process**




The refinement of DFDs continues until each bubble performs a single function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component

**2. Creating a control flow model**

For many types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As we have already noted, however, a large class of applications are driven by events rather than data, produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of control flow modeling in addition to data flow modeling

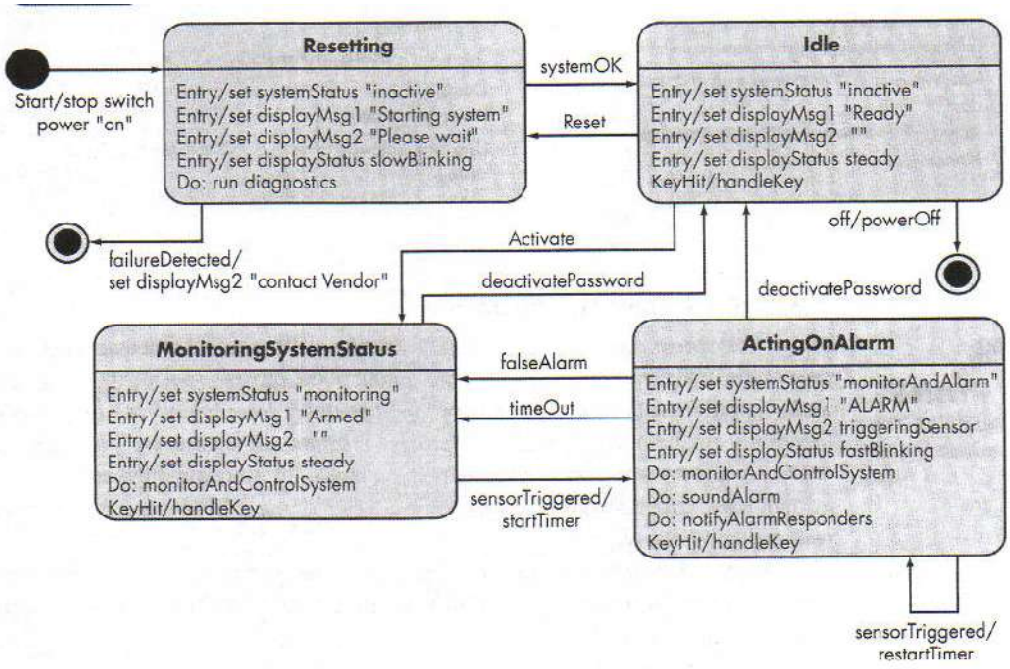
To select potential candidate events, the following guidelines are suggested:

- List all sensors that are read by the software
- List all interrupt conditions

- List all switches that are actuated by an operator
- List all data conditions
- Describe the behavior of a system by identifying its states, identify how each state is reached and define the transitions between states.
- Focus on possible omissions

**3. The Control Specification**

The Control Specification (CSPEC) represents the behavior of the system in two different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table-a combinatorial specification of behavior




**State diagram for safehome security function**

The diagram indicates how the system responds to events as it travels the four states defined at this level. By reviewing the state diagram, a software engineer can

determine the behavior of the system and, more importantly, can ascertain whether there are "holes" in the specified behavior

For example, the state diagram indicates the transitions from the idle state can occur if the system is reset activated or powered off. If the system is activated, a transition to the MonitoringSystemstatus state occurs, display messages are changed as shown, and the process MonitorAndControlSystem is invoked. Two transition occurs out of the MonitoringSystemStatus state- 1) when the system is deactivated a transition occurs back to the idle state, 2) when a sensor is triggered a transition to the during the review

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior

**4. The Process Specification**

The Process Specification (PSPEC) is used to describe allflow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language(PDL) description of the process algorithm, mathematical equations, tables, diagrams, or charts. By providing a PSPEC to accompany each bubble in the flow model, the software engineer creates a "mini-spec" that can serve as a guide for design of the software component that will implement the process

**Creating a Behavioral Model**

The behavioral model indicates how software will respond to external events. To create the model, the analyst must perform the following steps:

- Evaluate all use-cases to fully understand the sequence of interaction within the system.
- Identify events that drive the interaction sequence and understand how these events relate to specific classes
- Create a sequence for each use-case
- Build a state diagram for the system

- Review the behavioral model to verify accuracy and consistency

**1. Identifying Events with the Use-Case**

The use case represents a sequence of activities that involves actor and the system. An event occurs whenever the system and an actor exchange information. An actor should be identified for each event. The information that is exchanged should be noted and any conditions or constraints should be listed

In the context of the analysis model, the object, homeowner, transmits an event to the object control panel. The event might be password entered. The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use-case, while others have no impact on the flow of control.

For example, the event password entered does not explicitly change the control of the use-case, but the results of the event compare password will have an explicit impact on the information and control flow of the safehome software

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events

**2 .State Representations**

In the context of behavioral modeling, two different characterizations of states must be considered.

- The state of each class as the system performs its function
- The state of the system as observed from the outside as the system performs its function

The state of a class takes on both passive and active characteristics

**Passive state**

Passive state is simply the current status of all of an object's attributes

**Example**: The passive state of the class player would include the current position and orientation attributes of player as well as other features of player that are relevant to the game

**Active state**

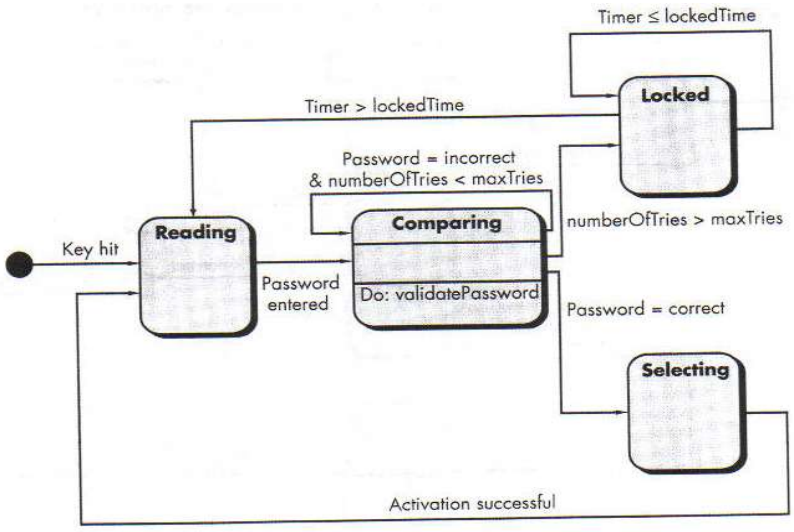
The active state of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class player might have the following active states: moving, at rest, injured, and being cured

An event must occur to force an object to make a transition from one active state to another. Two different behavioral representations are discussed

- The first indicates how an individual class changes state based on external events
- The second shows the behavior of the software as a function of time

**a) State diagrams for analysis classes**

One component of behavioral model is a UML state diagram that represents active state for each class and the events that cause changes between these active states.




**State diagram for the control panel class**

Each arrow represents a transition from one active state of a class to another. The labels shown for each arrow represent the event that triggers the transition. Although the active state model provides useful insight into the "life history" of a class, it is possible to specify additional information to provide more depth in understanding the behavior of a

class. In addition to specifying the event that causes the transition to occur, the analyst can specify guard and an action

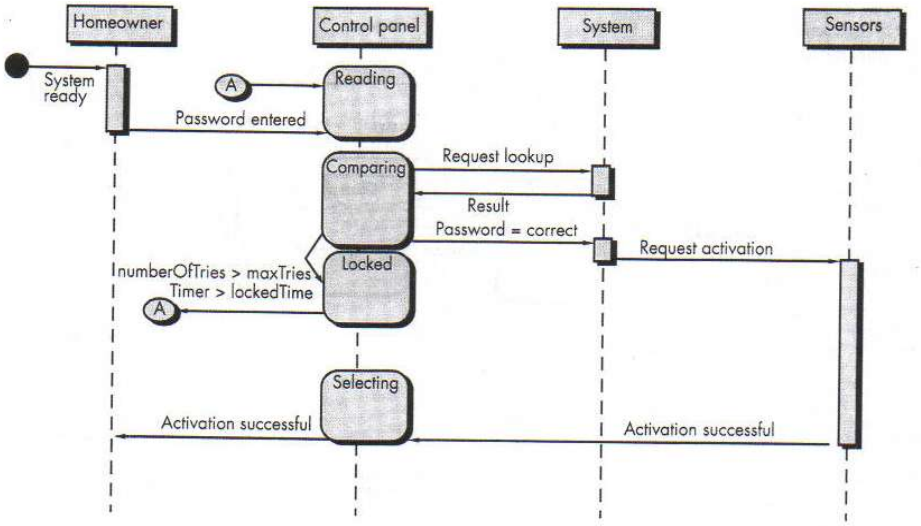
A guard is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing state" can be determined by examining the use case

An action occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations of an object. For example, the action connected to password entered event is an operation named validatepassword() that accesses a password object and performs a digit-by-digit comparison to validate the entered password

**b) Sequence diagrams**

The second type of behavioral representation called a sequence diagram in UML, indicates how events cause transitions from object to object once events have been Identified by examining use-case, the modeler creates a sequence diagram –a representation of how events cause flow from one object to another as a function of time.

Sequence diagram is a shorthand version of the use-case. It represents key classes and the events that cause behavior flow from class to class




Each of the arrow represent s an event and indicates how the event channels behavior between safehome objects. Time is measured vertically downward, and the narrow rectangles represent time spent in processing an activity. States may be shown along a vertical timeline

Once a complete sequence diagram has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events. This information is useful in the creation of an effective design for the system to be built

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE : 17CTU402** **UNIT II** **SEMESTER : IV**
**BATCH (2017-2020)**

Important Questions

1. Explain different types of Requirement Engineering Tasks.
2. How to initiate the Requirement Engineering Process.
3. Explain briefly about Eliciting Requirements.
4. Define Analysis Modeling approach.
5. Define Data Modeling concepts.
6. Explain about Flow- Oriented Modeling.
7. Explain about Active and Passive state of State Representation.

| | **Software Enginering (17CTU402)** | | **UNIT III** | |
|---|---|---|---|---|
| **S.No** | **Question** | **Option A** | **Option B** | **Option C** |
| 1 | There are _____ major phases to any design process | 2 | 3 | 4 |
| 2 | Diversification is the _____ of a repertoire of alternatives. | component | solution | acquisition |
| 3 | During _____, the designer chooses and combines appropriate | diversification | convergence | elimination |
| 4 | _____ and _____ combine intuition and judgement based on experience in building | elimination, convergence | creation, convergence | acquisition, creation |
| 5 | _____ can be traced to a customer's requirements and at the same time assessed | design | analysis | principles |
| 6 | The _____ must implement all of the explicit requirements contained in the | principles | testing | design |
| 7 | A _____ should exhibit an architectural structure that has been created using recognizable | principles | testing | component |
| 8 | A _____ is composed of components that exhibit good design characteristics. | principles | testing | component |
| 9 | A _____ can be implemented in an evolutionary fashion thereby | principles | testing | component |
| 10 | A _____ should be modular that is the software should be logically partitioned into elements that perform specific functions and sub functions. | design | principles | component |
| 11 | A _____ should contain distinct representations of data, architecture, interfaces, and | design | principles | component |

| | | | | |
|---|---|---|---|---|
| 12 | A _____ should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns. | design | principles | component |
| 13 | . A _____ should lead to interfaces that reduce the complexity of | design | principles | component |
| 14 | A _____ should be derived using a repeatable method that is driven by information obtained during | principles | component | design |
| 15 | The software _____ process encourages good design through the application of fundamental design principles, systematic methodology and thorough review. | principles | component | design |
| 16 | The _____ must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software. | principles | component | design |
| 17 | The _____ should provide a complete picture of the software addressing the data, functional and behavioral domains from an implementation perspective. | principles | component | design |
| 18 | The evolution of software _____ is a continuing process that has spanned the past four decades. | principles | component | design |
| 19 | Procedural aspects of design definition evolved into a philosophy called _____. | top down programming | bottom up programming | structured programming |
| 20 | The design process should not suffer from _____. | analysis | tunnel vision | conceptual errors |

| | | | | |
|---|---|---|---|---|
| 21 | The design should be _____ to the analysis model. | consistent | related | traceable |
| 22 | The design should not _____ the wheel. | minimize | maximize | integrate |
| 23 | The design should _____ the intellectual | maximize | minimize | integrate |
| 24 | . The _____ is represented at a high level of abstraction | specification | analysis | quality |
| 25 | The design should exhibit _____ and integration. | uniformity | analysis | quality |
| 26 | The design should be _____ to accommodate change. | reviewed | analysed | assessed |
| 27 | The design should be _____ to degrade gently, even when aberrant data, events, or operating | reviewed | analysed | assessed |
| 28 | Design is not _____' coding is not design | coding | analysis | review |
| 29 | Design is not coding, _____ is not design. | coding | analysis | review |
| 30 | The design should be _____ for quality as it is being created not after the fact. | reviewed | assessed | structured |
| 31 | The design should be _____ to minimize conceptual errors. | reviewed | assessed | structured |
| 32 | Software design is both a _____ and a model. | model | process | data |
| 33 | _____ is the only way that we can accurately translate a customer's requirements into a finished software product or system. | specification | design | data |

| | | | | |
|---|---|---|---|---|
| 34 | The design _____ is the equivalent of an architect's plan for a house. | analysis | process | model |
| 35 | At the highest level of _____, a solution is stated in broad terms, using the language of the problem environment. | refinement | modularity | abstraction |
| 36 | . A _____ is a named sequence of instructions that has a specific and limited function. | procedural abstraction | data abstraction | control abstraction |
| 37 | A _____ is a named collection of data that describes a data object. | procedural abstraction | data abstraction | control abstraction |
| 38 | _____ implies a program control mechanism without specifying internal detail. | procedural abstraction | data abstraction | control abstraction |
| 39 | _____ is used to coordinate activities in an operating system. | synchronization semaphore | control abstraction | data abstraction |
| 40 | _____ is a top down design strategy originally proposed by Niklaus Wirth. | stepwise refinement | control abstraction | data abstraction |
| 41 | The designer's goal is to produce a model or representation of a _____ that will later be built | component | entity | data |
| 42 | The second phase of any design process is the gradual _____ of all but one particular configuration of components, and thus the creation of the final product. | acquisition | addition | elimination |
| 43 | Design begins with the _____ model. | data | requirements | specification |
| 44 | Software design methodologies lack the _____ that are normally associated with more classical engineering design disciplines. | depth | flexibility | quantitative nature |

| | | | | |
|---|---|---|---|---|
| 45 | Software requirements, manifested by the _____ models, feed the design task. | data | functional | behavioral |
| 46 | _____ is the place where quality is fostered in software engineering | model | data | design |
| 47 | _____ provides us with representations of software that can be assessed for quality. | design | specification | data |
| 48 | Procedural aspects of design definition evolved into a philosophy called _____. | procedural programming | object oriented programming | structured programming |
| 49 | Meyer defines _____ criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system. | 2 | 3 | 4 |
| 50 | . If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution. This is called _____. | modular decomposability | modular composability | modular understandability |
| 51 | If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel. This is called _____. | modular decomposability | modular composability | modular understandability |
| 52 | If a module can be understood as a stand alone unit (without reference to other modules), it will be easier to build and easier to change. This is called _____. | modular decomposability | modular composability | modular understandability |

| | | | | |
|---|---|---|---|---|
| 53 | If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized. This is called _____. | modular decomposability | modular composability | modular understandability |
| 54 | If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized. This is called _____. | modular protection | modular composability | modular understandability |
| 55 | The aspect of the architectural design representation defines the components of a system and the manner in which those components are packaged and interact with one another. This property is called _____. | extra functional property | structural property | families of related systems |
| 56 | _____ represent architecture as an organized collection of program components. | dynamic models | functional models | framework models |
| 57 | _____ increases the level of design abstraction by attempting to identity repeatable architectural design frameworks that are encountered in similar types of applications. | framework models | dynamic models | process models |
| 58 | _____ address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. | framework models | dynamic models | process models |
| 59 | _____ focus on the design of the business or technical process that the system must accommodate. | framework models | dynamic models | process models |

| | | | | |
|---|---|---|---|---|
| 60 | _____ can be used to represent the functional hierarchy of a system. | framework models | dynamic models | process models |

| | | | |
|---|---|---|---|
| **Option D** | | | **Answer** |
| 5 | | | 2 |
| knowledge | | | acquisition |
| creation | | | convergence |
| diversification and convergence | | | diversification and convergence |
| testing | | | design |
| component | | | design |
| design | | | design |
| design | | | design |
| design | | | design |
| testing | | | design |
| testing | | | design |

| | | | |
|---|---|---|---|
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| object oriented programming | | | structured programming |
| integrity | | | tunnel vision |

| | | | |
|---|---|---|---|
| relevant | | | traceable |
| reinvent | | | reinvent |
| analyse | | | minimize |
| design specification | | | design specification |
| review | | | uniformity |
| structured | | | structured |
| structured | | | structured |
| event | | | coding |
| event | | | coding |
| integrated | | | assessed |
| integrated | | | reviewed |
| function | | | process |
| prototype | | | design |

| | | | |
|---|---|---|---|
| function | | | model |
| continuity | | | abstraction |
| Process abstraction | | | procedural abstraction |
| Process abstraction | | | data abstraction |
| Process abstraction | | | control abstraction |
| procedural abstraction | | | synchronization semaphore |
| procedural abstraction | | | stepwise refinement |
| raw material | | | component |
| creation | | | elimination |
| code | | | requirements |
| all of the above | | | all of the above |

| | | | |
|---|---|---|---|
| all of the above | | | all of the above |
| specification | | | design |
| prototype | | | design |
| all of the above | | | structured programming |
| 5 | | | 5 |
| modular continuity | | | modular decomposability |
| modular continuity | | | modular composability |
| modular continuity | | | modular understandability |

| | | | |
|---|---|---|---|
| modular continuity | | | modular continuity |
| modular continuity | | | modular protection |
| operational property | | | structural property |
| structural models | | | structural models |
| functional models | | | framework models |
| functional models | | | dynamic models |
| functional models | | | process models |

| | | | |
|---|---|---|---|
| functional models | | | functional models |

| | **Software Enginering (17CTU402)** | | **UNIT III** | |
|---|---|---|---|---|
| **S.No** | **Question** | **Option A** | **Option B** | **Option C** |
| 1 | There are _____ major phases to any design process | 2 | 3 | 4 |
| 2 | Diversification is the _____ of a repertoire of alternatives. | component | solution | acquisition |
| 3 | During _____, the designer chooses and combines appropriate | diversification | convergence | elimination |
| 4 | _____ and _____ combine intuition and judgement based on experience in building | elimination, convergence | creation, convergence | acquisition, creation |
| 5 | _____ can be traced to a customer's requirements and at the same time assessed | design | analysis | principles |
| 6 | The _____ must implement all of the explicit requirements contained in the | principles | testing | design |
| 7 | A _____ should exhibit an architectural structure that has been created using recognizable | principles | testing | component |
| 8 | A _____ is composed of components that exhibit good design characteristics. | principles | testing | component |
| 9 | A _____ can be implemented in an evolutionary fashion thereby | principles | testing | component |
| 10 | A _____ should be modular that is the software should be logically partitioned into elements that perform specific functions and sub functions. | design | principles | component |
| 11 | A _____ should contain distinct representations of data, architecture, interfaces, and | design | principles | component |

| | | | | |
|---|---|---|---|---|
| 12 | A _____ should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns. | design | principles | component |
| 13 | . A _____ should lead to interfaces that reduce the complexity of | design | principles | component |
| 14 | A _____ should be derived using a repeatable method that is driven by information obtained during | principles | component | design |
| 15 | The software _____ process encourages good design through the application of fundamental design principles, systematic methodology and thorough review. | principles | component | design |
| 16 | The _____ must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software. | principles | component | design |
| 17 | The _____ should provide a complete picture of the software addressing the data, functional and behavioral domains from an implementation perspective. | principles | component | design |
| 18 | The evolution of software _____ is a continuing process that has spanned the past four decades. | principles | component | design |
| 19 | Procedural aspects of design definition evolved into a philosophy called _____. | top down programming | bottom up programming | structured programming |
| 20 | The design process should not suffer from _____. | analysis | tunnel vision | conceptual errors |

| | | | | |
|---|---|---|---|---|
| 21 | The design should be _____ to the analysis model. | consistent | related | traceable |
| 22 | The design should not _____ the wheel. | minimize | maximize | integrate |
| 23 | The design should _____ the intellectual | maximize | minimize | integrate |
| 24 | . The _____ is represented at a high level of abstraction | specification | analysis | quality |
| 25 | The design should exhibit _____ and integration. | uniformity | analysis | quality |
| 26 | The design should be _____ to accommodate change. | reviewed | analysed | assessed |
| 27 | The design should be _____ to degrade gently, even when aberrant data, events, or operating | reviewed | analysed | assessed |
| 28 | Design is not _____' coding is not design | coding | analysis | review |
| 29 | Design is not coding, _____ is not design. | coding | analysis | review |
| 30 | The design should be _____ for quality as it is being created not after the fact. | reviewed | assessed | structured |
| 31 | The design should be _____ to minimize conceptual errors. | reviewed | assessed | structured |
| 32 | Software design is both a _____ and a model. | model | process | data |
| 33 | _____ is the only way that we can accurately translate a customer's requirements into a finished software product or system. | specification | design | data |

| | | | | |
|---|---|---|---|---|
| 34 | The design _____ is the equivalent of an architect's plan for a house. | analysis | process | model |
| 35 | At the highest level of _____, a solution is stated in broad terms, using the language of the problem environment. | refinement | modularity | abstraction |
| 36 | . A _____ is a named sequence of instructions that has a specific and limited function. | procedural abstraction | data abstraction | control abstraction |
| 37 | A _____ is a named collection of data that describes a data object. | procedural abstraction | data abstraction | control abstraction |
| 38 | _____ implies a program control mechanism without specifying internal detail. | procedural abstraction | data abstraction | control abstraction |
| 39 | _____ is used to coordinate activities in an operating system. | synchronization semaphore | control abstraction | data abstraction |
| 40 | _____ is a top down design strategy originally proposed by Niklaus Wirth. | stepwise refinement | control abstraction | data abstraction |
| 41 | The designer's goal is to produce a model or representation of a _____ that will later be built | component | entity | data |
| 42 | The second phase of any design process is the gradual _____ of all but one particular configuration of components, and thus the creation of the final product. | acquisition | addition | elimination |
| 43 | Design begins with the _____ model. | data | requirements | specification |
| 44 | Software design methodologies lack the _____ that are normally associated with more classical engineering design disciplines. | depth | flexibility | quantitative nature |

| | | | | |
|---|---|---|---|---|
| 45 | Software requirements, manifested by the _____ models, feed the design task. | data | functional | behavioral |
| 46 | _____ is the place where quality is fostered in software engineering | model | data | design |
| 47 | _____ provides us with representations of software that can be assessed for quality. | design | specification | data |
| 48 | Procedural aspects of design definition evolved into a philosophy called _____. | procedural programming | object oriented programming | structured programming |
| 49 | Meyer defines _____ criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system. | 2 | 3 | 4 |
| 50 | . If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution. This is called _____. | modular decomposability | modular composability | modular understandability |
| 51 | If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel. This is called _____. | modular decomposability | modular composability | modular understandability |
| 52 | If a module can be understood as a stand alone unit (without reference to other modules), it will be easier to build and easier to change. This is called _____. | modular decomposability | modular composability | modular understandability |

| | | | | |
|---|---|---|---|---|
| 53 | If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized. This is called _____. | modular decomposability | modular composability | modular understandability |
| 54 | If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized. This is called _____. | modular protection | modular composability | modular understandability |
| 55 | The aspect of the architectural design representation defines the components of a system and the manner in which those components are packaged and interact with one another. This property is called _____. | extra functional property | structural property | families of related systems |
| 56 | _____ represent architecture as an organized collection of program components. | dynamic models | functional models | framework models |
| 57 | _____ increases the level of design abstraction by attempting to identity repeatable architectural design frameworks that are encountered in similar types of applications. | framework models | dynamic models | process models |
| 58 | _____ address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. | framework models | dynamic models | process models |
| 59 | _____ focus on the design of the business or technical process that the system must accommodate. | framework models | dynamic models | process models |

| | | | | |
|---|---|---|---|---|
| 60 | _____ can be used to represent the functional hierarchy of a system. | framework models | dynamic models | process models |

| | | | |
|---|---|---|---|
| **Option D** | | | **Answer** |
| 5 | | | 2 |
| knowledge | | | acquisition |
| creation | | | convergence |
| diversification and convergence | | | diversification and convergence |
| testing | | | design |
| component | | | design |
| design | | | design |
| design | | | design |
| design | | | design |
| testing | | | design |
| testing | | | design |

| | | | |
|---|---|---|---|
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| testing | | | design |
| object oriented programming | | | structured programming |
| integrity | | | tunnel vision |

| | | | |
|---|---|---|---|
| relevant | | | traceable |
| reinvent | | | reinvent |
| analyse | | | minimize |
| design specification | | | design specification |
| review | | | uniformity |
| structured | | | structured |
| structured | | | structured |
| event | | | coding |
| event | | | coding |
| integrated | | | assessed |
| integrated | | | reviewed |
| function | | | process |
| prototype | | | design |

| | | | |
|---|---|---|---|
| function | | | model |
| continuity | | | abstraction |
| Process abstraction | | | procedural abstraction |
| Process abstraction | | | data abstraction |
| Process abstraction | | | control abstraction |
| procedural abstraction | | | synchronization semaphore |
| procedural abstraction | | | stepwise refinement |
| raw material | | | component |
| creation | | | elimination |
| code | | | requirements |
| all of the above | | | all of the above |

| | | | |
|---|---|---|---|
| all of the above | | | all of the above |
| specification | | | design |
| prototype | | | design |
| all of the above | | | structured programming |
| 5 | | | 5 |
| modular continuity | | | modular decomposability |
| modular continuity | | | modular composability |
| modular continuity | | | modular understandability |

| | | | |
|---|---|---|---|
| modular continuity | | | modular continuity |
| modular continuity | | | modular protection |
| operational property | | | structural property |
| structural models | | | structural models |
| functional models | | | framework models |
| functional models | | | dynamic models |
| functional models | | | process models |

| | | | |
|---|---|---|---|
| functional models | | | functional models |

**UNIT-IV**

Design Engineering-Design Concepts, Architectural Design Elements, Software Architecture,Data Design at the Architectural Level and Component Level, Mapping of Data Flow into Software Architecture, Modeling Component Level Design

## Design Engineering

### 4.1 Design within the Context of Software Engineering

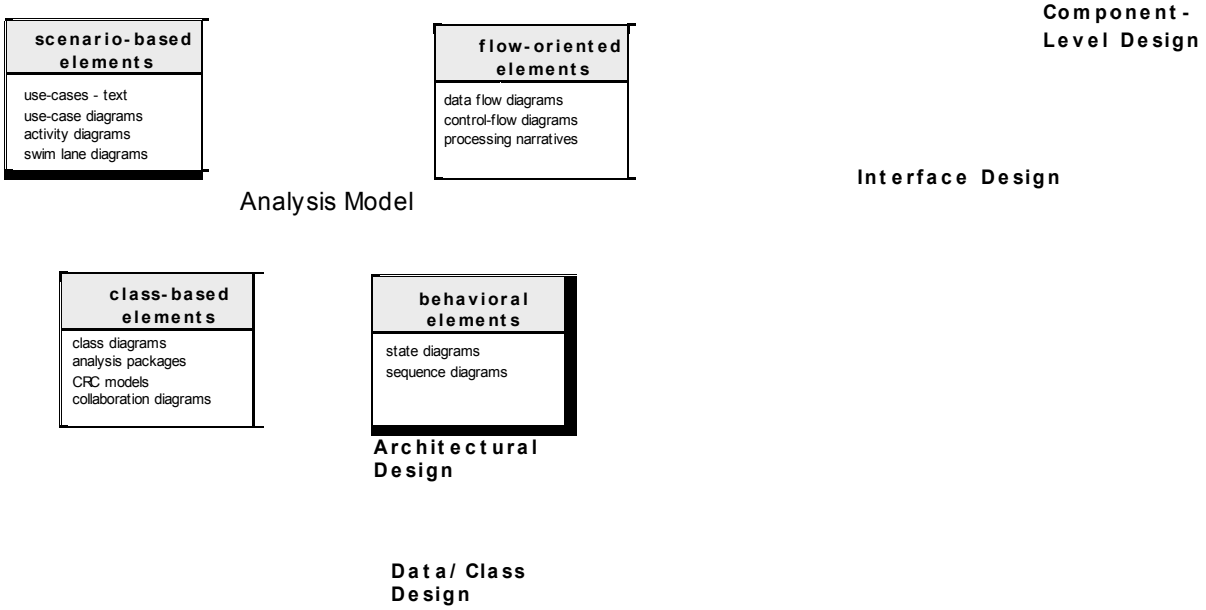
Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

The flow of information during software design is illustrated in Figure below. The analysis model, manifested by scenario-based, class-based, flow-oriented and behavioral elements, feed the design task.

The *architectural design* defines the relationship between more structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which the architectural design can be implemented.

The *architectural design* can be derived from the System Specs, the analysis model, and interaction of subsystems defined within the analysis model.

The *interface design* describes how the software communicates with systems that interpolate with it, and with humans who use it. An interface implies a flow of information (data, and or control) and a specific type of behavior.


scenario-based elements
use-cases - text
use-case diagrams
activity diagrams
swim lane diagrams
flow-oriented elements
data flow diagrams
control-flow diagrams
processing narratives
Component-Level Design
Interface Design
Analysis Model
class-based elements
class diagrams
analysis packages
CRC models
collaboration diagrams
behavioral elements
state diagrams
sequence diagrams
Architectural Design
Data/ Class Design

n

M
o
d
e
l

The *component-level design* transforms structural elements of the software architecture into a procedural description of software components

The importance of software design can be stated with a single word – *quality*. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system.

### 4.2 Design Process and Design Quality

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

Initially, the **blueprint** depicts a holistic view of software, i.e. the design is represented at a high-level of abstraction.

Throughout the design process, the quality of the evolving design is assessed with a series of formal technique reviews or design walkthroughs.

Three characteristics serve as a guide for the evaluation of a good design:

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

**Quality Guidelines**

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

1. A design should exhibit an architecture that:
   (1) Has been created using recognizable architectural styles or patterns,

   (2) Is composed of components that exhibit good design characteristics, and

   (3) Can be implemented in an evolutionary fashion

   a. For smaller systems, design can sometimes be developed linearly.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

**Quality Attributes**

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS. The FURPS quality attributes represent a target for all software design:

- ✓ *Functionality*: is assessed by evaluating the features set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ✓ *Usability*: is assessed by considering human factors, overall aesthetics, consistency, and documentation.

- ✓ *Reliability*: is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.
- ✓ *Performance*: is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- ✓ *Supportability*: combines the ability to extend the program extensibility, adaptability, serviceability ➔ maintainability. In addition, testability, compatibility, configurability, etc.

### 4.3 Design Concepts

This section discusses many significant design concepts (abstraction, refinement, modularity, architecture, patterns, refactoring, functional independence, information hiding, and OO design concepts).

#### 4.3.1 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.

As we move through different levels of abstraction, we work to create procedural and data abstractions. 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.

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g. **door type, swing direction, weight)**.

#### 4.3.2 Architecture

Software architecture alludes to the ―overall structure of the software and the ways in which the structure provides conceptual integrity for a system.‖

In its simplest from, architecture is the structure of organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Te goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which detailed design activities are constructed.

A set of architectural patterns enable a software engineer to reuse design-level concepts.

The architectural design can be represented using one or more of a number of different models.

*Structural models* represent architecture as an organized collection of program components.

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

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

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

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

Architectural design will be discussed in Chapter 10.

### 4.3.3 Patterns

A design pattern ―conveys the essence of a proven design solution to a recurring problem within a certain context amidst computing concerns.‖

A design pattern describes a design structure that solves a particular design problem within a specific context and amid ―forces‖ that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

1. whether the pattern is 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.

### 4.3.4 Modularity

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

Monolithic software (large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

It is the compartmentalization of data and function. It is easier to solve a complex problem when you break it into manageable pieces. ―Divide-and-conquer‖

Don't over-modularize. The simplicity of each small module will be overshadowed by the complexity of integration ―Cost‖.

### 4.3.5 Information Hiding

It is about controlled interfaces. Modules should be specified and design so that information (algorithm 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 by a set of independent modules that communicate with one another only that information necessary to achieve software function.

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

### 4.3.6 Functional Independence

The concept of *functional Independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.

Functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: **cohesion** and **coupling.**

**Cohesion** is an indication of the relative functional strength of a module.

**Coupling** is an indication of the relative interdependence among modules.

A cohesive module should do just one thing.

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world ―lowest possible‖.

### 4.3.7 Refinement

It is the elaboration of detail for all abstractions. It is a top down strategy.

A program is developed by successfully refining levels of procedural detail.

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

We begin with a statement of function or data that is defined at a high level of abstraction.

The statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data.

Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction enables a designer to specify procedure and data and yet suppress low-level details.

Refinement helps the designer to reveal low-level details as design progresses.

Refinement causes the designer to elaborate on the original statement, providing more and more finement —elaboration occurs.

**Procedural Abstraction**

**"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system."**

- Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods
- Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

**Patterns**

***Design Pattern Template***

***Pattern name***—describes the essence of the pattern in a short but expressive name

***Intent***—describes the pattern and what it does

***Also-known-as***—lists any synonyms for the pattern

***Motivation***—provides an example of the problem

***Applicability***—notes specific design situations in which the pattern is applicable

***Structure***—describes the classes that are required to implement the pattern

***Participants***—describes the responsibilities of the classes that are required to implement the pattern

***Collaborations***—describes how the participants collaborate to carry out their responsibilities

***Consequences***—describes the ―design forces‖ that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

***Related patterns***—cross-references related design patterns

**Modular Design**

*easier to build, easier to change, easier to fix …*

**Modularity: Trade-offs**

*What is the "right" number of modules for a specific software design?*

**Information Hiding**

**Why Information Hiding?**

- Reduces the likelihood of ―side effects‖
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

**Stepwise Refinement**

**Functional Independence**

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

**Refactoring**

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.‖
- When software is re-factored, 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.

**Design Concepts**

- Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a super-class is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

### 4.3.9 Design classes

As the design model evolves, the software team must define a set of *design classes* that refines the analysis classes and creates a new set of design classes.

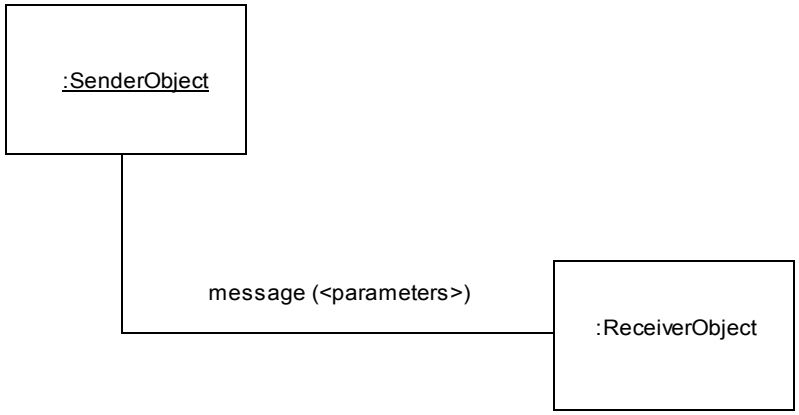
Five different classes' types are shown below:

1. *User Interface classes*: define all abstractions that are necessary for HCI.
2. *Business domain classes*: are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.
3. *Process classes:* implement lower-level business abstractions required to fully manage the business domain classes.
4. *Persistent classes:* represent data stores that will persist beyond the execution of the software.
5. *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.

**Inheritance (Example)**

- Design options:
  - The class can be designed and built from scratch. That is, inheritance is not used.
  - The class hierarchy can be searched to determine if a class higher in the hierarchy (a super-class) contains most of the required attributes and operations. The new class inherits from the super-class and additions may then be added, as required.
  - The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
  - Characteristics of an existing class can be overridden and different versions of attributes or operations are implemented for the new class.

**Messages**




**Polymorphism**

Conventional approach …

**case of graphtype:**

**if graphtype = linegraph then DrawLineGraph (data);**

**if graphtype = piechart then DrawPieChart (data);**

**if graphtype = histogram then DrawHisto (data);**

**if graphtype = kiviat then DrawKiviat (data);**

**end case;**

All of the graphs become subclasses of a general class called graph. Using a concept called overloading [TAY90], each subclass defines an operation called *draw*. An object can send a

*draw* message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own *draw* operation to create the appropriate graph.

**Architectural design elements**

- The architecture design elements provides us overall view of the system.
- The architectural design element is generally represented as a set of interconnected subsystem that are derived from analysis packages in the requirement model. **The architecture model is derived from following sources:**
- The information about the application domain to built the software.
- Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.
- The architectural style and pattern as per availability. **3. Interface design elements**
- The interface design elements for software represents the information flow within it and out of the system.
- They communicate between the components defined as part of architecture. **Following are the important elements of the interface design:**

1. The user interface
2. The external interface to the other systems, networks etc. 3. The internal interface between various components.

**4. Component level diagram elements**

- The component level design for software is similar to the set of detailed specification of each room in a house.
- The component level design for the software completely describes the internal details of the each software component.
- The processing of data structure occurs in a component and an interface which allows all the component operations.

- In a context of object-oriented software engineering, a component shown in a UML diagram. • The UML diagram is used to represent the processing logic.

**5. Deployment level design elements**

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

**Software Architecture Introduction**

- The concept of software architecture is similar to the architecture of building. • The architecture is not an operational software.
- The software architecture focuses on the role of software components.
- Software components consist of a simple program module or an object oriented class in an architectural design.
- The architecture design extended and it consists of the database and the middleware that allows the configuration of a network of clients and servers.

Importance of software architecture

**Following are the reasons for the importance of software architecture.**

1. The representation of software architecture allows the communication between all stakeholder and the developer.
2. The architecture focuses on the early design decisions that impact on all software engineering work and it is the ultimate success of the system.
3. The software architecture composes a small and intellectually graspable model.
4. This model helps the system for integrating the components using which the components are work together.

The architectural style

- The architectural style is a transformation and it is applied to the design of an entire system.
- The main aim of architectural style is to build a structure for all components of the system.
- An architecture of the system is redefined by using the architectural style.
- An architectural pattern such as architectural style introduces a transformation on the design of an architecture.
- The software is constructed for computer based system and it shows one of the architectural style from many of style.

**The design categories of architectural styles includes:**

1. A set of components such as database, computational modules which perform the function required by the system.
2. A set of connectors that allows the communication, coordination and cooperation between the omponents.
3. The constraints which define the integration of components to form the system.
4. Semantic model allows a designer to understand the overall properties of a system by using analysis of elements.

Architectural design

- The architectural design starts then the developed software is put into the context.
- The information is obtained from the requirement model and other information collect during the requirement engineering.

**Representing the system in context**

All the following entities communicates with the target system through the interface that is small rectangles shown in above figure.

**Superordinate system**

These system use the target system like a part of some higher-level processing scheme.

**Subordinate system**
This systems is used by the target system and provide the data mandatory to complete target system functionality.

**Peer-level system**
These system interact on peer-to-peer basis means the information is consumed by the target system and the peers.

**Actors**
These are the entities like people, device which interact with the target system by consuming information that is mandatory for requisite processing.

Defining Archetype

- An archetype is a class or pattern which represents a core abstraction i.e critical to implement or design for the target system.
- A small set of archetype is needed to design even the systems are relatively complex.
- The target system consists of archetype that represent the stable elements of the architecture. • Archetype is instantiated in many different forms based on the behavior of the system.
- In many cases, the archetype is obtained by examining the analysis of classes defined as a part of the requirement model.

An Architecture Trade-off Analysis Method (ATAM)

ATAM was developed by the Software Engineering Institute (SEI) which started an iterative evaluation process for software architecture.

**The design analysis activities which are executed iteratively that are as follows:**

1. **Collect framework**

Collect framework developed a set of use cases that represent the system according to user point of view.

**2. Obtained requirement, Constraints, description of the environment.**
These types of information are found as a part of requirement engineering and is used to verify all the stakeholders are addressed properly.

**3. Describe the architectural pattern**
The architectural patterns are described using an architectural views which are as follows:

**Module view:** This view is for the analysis of assignment work with the components and the degree in which abstraction or information hiding is achieved

**Process view:** This view is for the analysis of the software or system performance.

**Data flow view:** This view analyzes the level and check whether functional requirements are met to the architecture.

**4. Consider the quality attribute in segregation**
The quality attributes for architectural design consist of reliability, performance, security, maintainability, flexibility, testability, portability, re-usability etc.

**5. Identify the quality attributes sensitivity**

- The sensitivity of quality attributes achieved by making the small changes in the architecture and find the sensitivity of the quality attribute which affects the performance.
- The attributes affected by the variation in the architecture are known as sensitivity points.

**Data Design at the Architectural Level and Component Level**
The data design action translates data defined as part of the analysis model into data structures at the software component level and. When necessary into a database architecture at the application level.

**a) Data Design at the Architectural Level**

The challenge in data design is to extract useful information from this data environment, particularly when the information desired is cross-functional.

To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in database (KDD) , that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a data warehouse, adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day –to-day application but encompasses all data used by a business.

**b) Data Design at the Component Level**

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

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structure and the operations to be performed on each should be identified.
3. A mechanism for defining the content of each data object should be established and used to define both data and the operation applied it.

4. Low-level design decision should be known only to those modules that must make direct use of the data contained within the structure.
5. The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

### Mapping Data Flow into Software Architecture

This section describes the general process of mapping requirements into software architectures during the structured design process. The technique described in this chapter is based on analysis of the data flow diagram discussed in Chapter 8.

### An Architectural Design Method

*customer requirements*

four bedrooms, three baths, lots of glass…

### Deriving Program Architecture

### Partitioning the Architecture

horizontal" and "vertical" partitioning are required

### Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions

**Vertical Partitioning:**
**Factoring**

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture

**Why Partitioned Architecture?**

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

- objective: to derive a program architecture that is partitioned
- approach:
  - the DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- notation: structure chart

**Flow Characteristics**

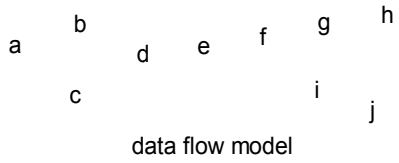
**General Mapping Approach**

Isolate incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center.

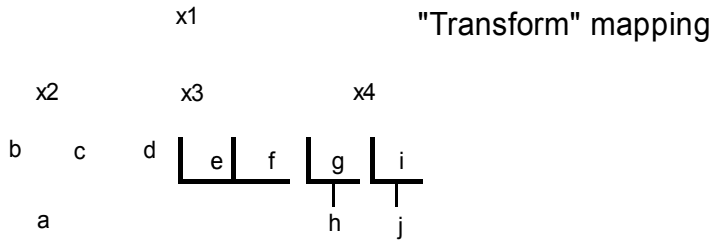
Working from the boundary outward, map DFD transforms into corresponding modules.

Add control modules as required.

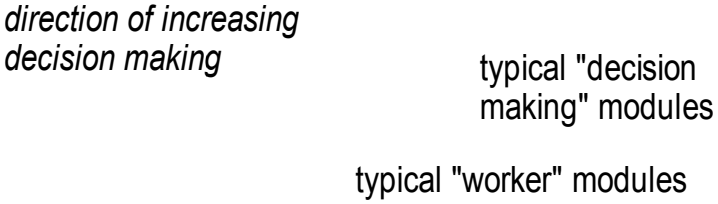
Refine the resultant program structure using effective modularity concepts.




data flow model



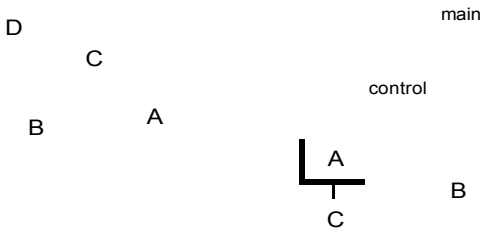

**Factoring**



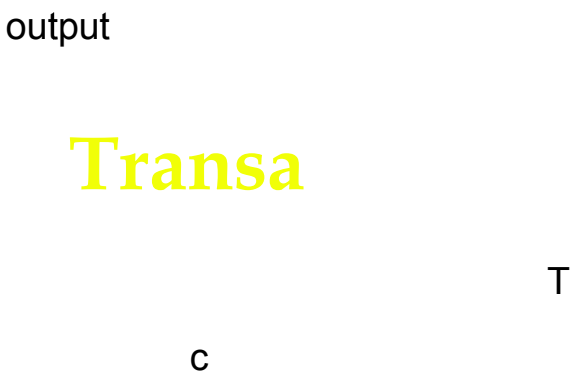

**First Level Factoring**

main

## Second Level Mapping




**Transaction Flow**




**Refining the Analysis Model**

1. Write an English language processing narrative for the level 01 flow model
2. Apply noun/verb parse to isolate processes, data items, store and entities
3. Develop level 02 and 03 flow models
4. Create corresponding data dictionary entries
5. Refine flow models as appropriate

## Modeling Component Level Design

Overview

The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design. Component-level design occurs after the data and architectural designs are established. The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built. The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation. Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps.

Component Definitions

- Component is a modular, deployable, replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- Object-oriented view is that component contains a set of collaborating classes
    - Each elaborated class includes all attributes and operations relevant to its implementation
    - All interfaces communication and collaboration with other design classes are also defined
    - Analysis classes and infrastructure classes serve as the basis for object-oriented elaboration
- Traditional view is that a component (or module) reside in the software and serves one of three roles
    - Control components coordinate invocation of all other problem domain components
    - Problem domain components implement a function required by the customer
    - Infrastructure components are responsible for functions needed to support the processing required in a domain application
    - The analysis model data flow diagram is mapped into a module hierarchy as the starting point for the component derivation

- Process-Related view emphasizes building systems out of existing components chosen from a catalog of reusable components as a means of populating the architecture

Class-based Component Design

- Focuses on the elaboration of domain specific analysis classes and the definition of infrastructure classes
- Detailed description of class attributes, operations, and interfaces is required prior to beginning construction activities

Class-based Component Design Principles

- Open-Closed Principle (OCP) – class should be open for extension but closed for modification
- Liskov Substitution Principle (LSP) – subclasses should be substitutable for their base classes
- Dependency Inversion Principle (DIP) – depend on abstractions, do not depend on concretions
- Interface Segregation Principle (ISP) – many client specific interfaces are better than one general purpose interface
- Release Reuse Equivalency Principle (REP) – the granule of reuse is the granule of release
- Common Closure Principle (CCP) – classes that change together belong together
- Common Reuse Principle (CRP) – Classes that can't be used together should not be grouped together

Component-Level Design Guidelines

- Components
  - Establish naming conventions in during architectural modeling
  - Architectural component names should have meaning to stakeholders
  - Infrastructure component names should reflect implementation specific meanings
  - Use of stereotypes may help identify the nature of components
- Interfaces
  - Use lollipop representation rather than formal UML box and arrow notation
  - For consistency interfaces should flow from the left-hand side of the component box
  - Show only the interfaces relevant to the component under construction
- Dependencies and Inheritance
  - For improved readability model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)

- Component interdependencies should be represented by interfaces rather that component to component dependencies

**Cohesion (lowest to highest)**

- Utility cohesion – components grouped within the same category but are otherwise unrelated
- Temporal cohesion – operations are performed to reflect a specific behavior or state
- Procedural cohesion – components grouped to allow one be invoked immediately after the preceding one was invoked with or without passing data
- Communicational cohesion –operations required same data are grouped in same class
- Sequential cohesion – components grouped to allow input to be passed from first to second and so on
- Layer cohesion – exhibited by package components when a higher level layer accesses the services of a lower layer, but lower level layers do not access higher level layer services
- Functional cohesion – module performs one and only one function

**Coupling**

- Content coupling – occurs when one component surreptitiously modifies internal data in another component
- Common coupling – occurs when several components make use of a global variable
- Control coupling – occurs when one component passes control flags as arguments to another
- Stamp coupling – occurs when parts of larger data structures are passed between components
- Data coupling – occurs when long strings of arguments are passed between components
- Routine call coupling – occurs when one operator invokes another
- Type use coupling – occurs when one component uses a data type defined in another
- Inclusion or import coupling – occurs when one component imports a package or uses the content of another
- External coupling – occurs when a components communications or collaborates with infrastructure components (e.g. database)

Conducting Component-Level Design

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
    a. Specify message details when classes or components collaborate.
    b. Identify appropriate interfaces for each component.
    c. Elaborate attributes and define data types and data structures required to implement them.
    d. Describe processing flow within each operation in detail.
4. Identify persistent data sources (databases and files) and identify the classes required to manage them.
5. Develop and elaborate behavioral representations for each class or component.
6. Elaborate deployment diagrams to provide additional implementation detail.
7. Refactor every component-level diagram representation and consider alternatives.

WebApp Component-Level Design

- Boundary between content and function often blurred
- WebApp component is defined is either a:
    - well-defined cohesive function manipulates content or provides computational or data processing for an end- user or
    - cohesive package of content and functionality that provides the end-user with some required capability

WebApp Component-Level Content Design

- Focuses on content objects and the manner in which they may be packaged for presentation to the end-user
- As the WebApp size increases so does the need for formal representations and easy content reference and manipulation
- For highly dynamic content a clear structural model incorporating content components should be established

WepApp Component-Level Functional Design

- WebApps provide sophisticated processing functions

- perform dynamic processing to create content and navigational capability
- provide business domain appropriate computation or data processing
- provide database query and access
- establish interfaces with external corporate systems

- WebApp functionality is delivered as a series of components developed in parallel
- During architectural design WebApp content and functionality are combined to create a functional architecture
- The functional architecture is a representation of the functional domain of the WebApp and describes how the components interact with each other

Traditional Component-Level Design

- Each block of code has a single entry at the top
- Each block of code has a single exit at the bottom
- Only three control structures are required: sequence, condition (if-then-else), and repetition (looping)
- Reduces program complexity by enhancing readability, testability, and maintainability

Design Notation

- Graphical
  - UML activity diagrams
  - Flowcharts – arrows for flow of control, diamonds for decisions, rectangles for processes
- Tabular
  - Decision table – subsets of system conditions and actions are associated with each other to define the rules for processing inputs and events
- Program Design Language (PDL)
  - Structured English or pseudocode used to describe processing details
  - Fixed syntax with keywords providing for representation of all structured constructs, data declarations, and module definitions
  - Free syntax of natural language for describing processing features
  - Data declaration facilities for simple and complex data structures
  - Subprogram definition and invocation facilities

Component-Based Development

- CBSE is a process that emphasizes the design and construction of computer-based systems from a catalog of reusable software components
- CBSE is a time and cost effective
- Requires software engineers to reuse rather than reinvent
- Management can be convinced to incur the additional expense required to create reusable components by amortizing the cost over multiple projects
- Libraries can be created to make reusable components easy to locate and easy to incorporate them in new systems

**Possible Questions**

**Part – B (2 Mark)**

1. Define abstraction.
2. Differentiate between refinement and refactoring
3. Write the difference between transform flow and transaction flow
4. What is transform mapping?
5. Define transaction mapping.

**Part – C (6 Mark)**

1. Explain in detail the process of data design at
    a. Architectural level ii) Component level
2. Write in detail the approach used to design class based components
3. Discuss in detail about the Architectural components of software.
4. Write short notes on
    a. Transform mapping ii) Transaction mapping
5. Write short notes on the following design concepts
    a. Information hiding ii) Refinement iii) Refactoring
6. Describe in detail the procedure to refine an architecture into components.
7. Write short notes on the following design concepts
    a. Abstraction ii) Architecture iii) Modularity
8. Write in detail the approach used to design conventional components
9. Explain in detail about design process and design quality
10. Write short notes on
    a. Transform flow ii) Transaction flow

| | **Software Enginering (17CTU402)** | | **UNIT IV** | |
|---|---|---|---|---|
| **S.No** | **Question** | **Option A** | **Option B** | **Option C** |
| 1 | Interface design focuses on _____ areas of concern. | 2 | 3 | 4 |
| 2 | . Frustration and _____ are part of daily life for many users of computerized information system | sadness | happiness | enjoyment |
| 3 | _____ creates effective communication medium between a human and a computer. | user interface design | architectural design | code design |
| 4 | _____ identifies interface objects and actions and then creates a screen layout that form the basis for a user interface prototype. | design | coding | testing |
| 5 | _____ begins with the identification of user, task and environmental requirements. | user interface design | architectural design | code design |
| 6 | There are _____ golden rules. | 2 | 3 | 4 |
| 7 | We should define interaction modes in a way that does not force a user into unnecessary or undesired actions. | interaction modes | interface constraints | design principles |
| 8 | We should provide _____ interaction. | rigid | flexible | encouraging |
| 9 | We should design for direct interaction with _____ that appear on the screen | code | class | objects |

| | | | | |
|---|---|---|---|---|
| 10 | We should hide technical _____ from the casual user | reactions | actions | internals |
| 11 | We should streamline _____ as skill levels advance and allow the interaction to be customized. | internals | interaction | actions |
| 12 | . We should allow user interaction to be _____ and undoable | interruptible | flexible | rigid |
| 13 | We should allow user interaction to interruptible and _____. | undoable | flexible | rigid |
| 14 | We should define shortcuts that are _____. | encouraging | intuitive | default |
| 15 | We should define _____ that are intuitive. | shortcuts | broad area | interruptible actions |
| 16 | We should disclose information in a _____ fashion. | open | progressive | streamline |
| 17 | The visual layout of the _____ should be based on a real world metaphor. | interaction modes | interface | design |
| 18 | The interface should present and acquire _____ in a consistent fashion. | information | task | knowledge |
| 19 | The interface should present and acquire information in a _____ fashion. | consistent | inconsistent | rigid |
| 20 | A _____ of the entire system incorporates data, architectural interface, and procedural representations of the software | data model | design model | user model |
| 21 | The software engineer creates a _____. | design model | data model | interface model |

| | | | | |
|---|---|---|---|---|
| 22 | The end user develops a mental image that is often called the _____. | design model | user model | data model |
| 23 | The implementers of the system create a _____. | design model | system image | data model |
| 24 | Users are categorized into _____ types. | 2 | 3 | 4 |
| 25 | Users with no syntactic knowledge of the system and little semantic knowledge of the application or computer usage are called _____. | knowledgeable intermittent users | knowledgeable frequent users | novices |
| 26 | Users with reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface are called _____. | novices | knowledgeable, intermittent users | knowledgeable, frequent users |
| 27 | Users with good semantic and syntactic knowledge that often leads to the “power-user syndrome” are called _____. | novices | knowledgeable, intermittent users | knowledgeable, frequent users |
| 28 | Individuals who look for shortcuts and abbreviated modes of interaction are called _____. | novices | knowledgeable, intermittent users | knowledgeable, frequent users |
| 29 | The _____ is the image of the system that end-users carry in their heads. | user’s model | data model | design model |
| 30 | Stepwise elaboration is called _____. | functional decomposition | data abstraction | modularity |

| | | | | |
|---|---|---|---|---|
| 31 | _____ is the only way that we can accurately translate a customer's requirements into a finished software product or system. | specification | design | data |
| 32 | Validation focuses on _____ criteria. | 2 | 3 | 4 |
| 33 | Task analysis can be applied in _____ ways. | 2 | 3 | 4 |
| 34 | Task analysis for interface design used _____ approach. | object oriented approach | top down approach | bottom up approach |
| 35 | The overall approach to task analysis, a human engineer must first _____ and classify tasks. | discuss | define | describe |
| 36 | There are _____ steps in interface design activities. | 4 | 5 | 6 |
| 37 | _____ refers to the deviation from average time. | system response time | variability | system mean time |
| 38 | System response time has _____ important characteristics. | | 3 | 4 |
| 39 | A _____ is designed into the software from the beginning. | integrated help facility | system response time | variability |
| 40 | Component level design also called _____. | procedural abstraction | procedural design | stepwise refinement |
| 41 | _____ must be translated into operational software | data | architectural | interface design |
| 42 | A _____ performs component level design. | user | top level management | software engineer |

| | | | | |
|---|---|---|---|---|
| 43 | The _____ represents the software in a way that allows one to review the details of the design for correctness and consistency with earlier design representations. | component level design | procedural design | data design |
| 44 | Design, representations of data, architecture, and interfaces form the foundation for _____. | procedural design | component level design | data design |
| 45 | _____ notation is used to represent the design. | graphical | tabular | text-based |
| 46 | Any program, regardless of application area or technical complexity, can be designed and implemented using only the _____ structured constructs. | 2 | 3 | 4 |
| 47 | A box in a flowchart is used to indicate a _____. | processing step | logical condition | flow of control |
| 48 | A diamond in a flowchart is used to indicate a _____. | processing step | logical condition | flow of control |
| 49 | The arrows in a flowchart is used to indicate a _____. | processing step | logical condition | flow of control |
| 50 | A picture is worth a _____ words. | 100 | 1000 | 10000 |
| 51 | The following construct is fundamental to structured programming. | sequence | condition | repetition |
| 52 | _____ implements processing steps that are essential in the specification of any algorithm. | sequence | condition | repetition |

| | | | | |
|---|---|---|---|---|
| 53 | _____ provides the facility for selected processing steps that are essential in the specification of any algorithm | sequence | condition | repetition |
| 54 | _____ allows for looping. | sequence | condition | repetition |
| 55 | Another graphical design tool, the _____ evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. | box diagram | flowchart | transition diagram |
| 56 | PDL is the abbreviation of _____. | Process Design Language | Program Design Language | Program Document Language |
| 57 | A design language should have the _____ characters. | 2 | 3 | 4 |
| 58 | Design notation should support the development of modular software and provide a means for interface specification. This attribute of design notation is called _____. | modularity | simplicity | ease of editing |
| 59 | Design notation should be relatively simple to learn, relatively easy to use, and generally easy to read. This attribute of the design notation is called _____. | modularity | simplicity | ) ease of editing |
| 60 | The procedural design may require modification as the software process proceeds. The ease with which a design representation can be edited can help facilitate each software engineering task is called _____. | modularity | simplicity | ease of editing |

| | | | |
|---|---|---|---|
| **Option D** | | | **Answer** |
| 5 | | | 3 |
| anxiety | | | anxiety |
| procedure design | | | user interface design |
| analysis | | | design |
| procedure design | | | user interface design |
| 5 | | | 3 |
| design analysis | | | interaction modes |
| enthusiastic | | | flexible |
| user | | | objects |

| | | | |
|---|---|---|---|
| interactions | | | internals |
| reactions | | | interaction |
| encouraging | | | interruptible |
| encouraging | | | undoable |
| past actions | | | intuitive |
| interactions | | | shortcuts |
| flexible | | | progressive |
| structure | | | interface |
| idea | | | information |
| flexible | | | consistent |
| system image | | | design model |
| system image | | | design model |

| | | | |
|---|---|---|---|
| system image | | | user model |
| user model | | | system image |
| 5 | | | 3 |
| all of the above | | | novices |
| all of the above | | | knowledgeable, intermittent users |
| all of the above | | | knowledgeable, frequent users |
| Testers | | | knowledgeable, frequent users |
| system image | | | user’s model |
| modular protection | | | functional decomposition |

| | | | |
|---|---|---|---|
| prototype | | | design |
| 5 | | | 2 |
| 5 | | | 3 |
| all of the above | | | object oriented approach |
| list | | | define |
| 7 | | | 7 |
| all of the above | | | variability |
| 5 | | | 2 |
| all of the above | | | integrated help facility |
| decomposition | | | procedural design |
| all of the above | | | all of the above |
| middle level management | | | software engineer |

| | | | |
|---|---|---|---|
| data design | | | component level design |
| code design | | | component level design |
| all of the above | | | graphical |
| 5 | | | 3 |
| start | | | processing step |
| start | | | logical condition |
| start | | | flow of control |
| 100000 | | | 1000 |
| all of the above | | | all of the above |
| selection | | | sequence |

| | | | |
|---|---|---|---|
| selection | | | condition |
| selection | | | repetition |
| decision table | | | box diagram |
| Program Document Language | | | Program Design Language |
| 5 | | | 4 |
| maintainability | | | modularity |
| maintainability | | | simplicity |
| maintainability | | | ease of editing |

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE : 17CTU402** **UNIT V** **SEMESTER : IV**
**BATCH (2017-2020)**

**UNIT-V**

**Testing Strategies & Tactics:** Software Testing Fundamentals, Strategic Approach to Software Testing, Test Strategies for Conventional Software, Validation Testing, System testing Black-Box Testing, White-Box Testing and their type, Basis Path Testing

## SOFTWARE TESTING FUNDAMENTALS:

- Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.
- Software engineers are by their nature constructive people.
- Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.
- Beizer describes this situation effectively when he states: There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt.

### Testing Objectives

Glen Myers states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software.

Also testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.

In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.

But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

**Testing Principles**

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis [DAV95] suggests a set of testing principles.

**All tests should be traceable to customer requirements**.

The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

**Tests should be planned long before testing begins**.

Test planning can begin as soon as the requirements model is complete.

Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

**The Pareto principle applies to software testing**.

Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

**Testing should begin "in the small" and progress toward testing "in the large."**

The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

**Exhaustive testing is not possible**

The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

**To be most effective, testing should be conducted by an independent third party.**

**Testability**

- Software testability is simply how easily a computer program can be tested.
- Since testing is so profoundly difficult, it pays to know what can be done to streamline it.
- Sometimes programmers are willing to do things that will help the testing process and a checklist of possible design points, features, etc., can be useful in negotiating with them.
- "Testability" occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult.
  The **checklist** that follows provides a set of characteristics that lead to testable software.

**Operability**. "The better it works, the more efficiently it can be tested."

• The system has few bugs (bugs add analysis and reporting overhead to the test process).

• No bugs block the execution of tests.

• The product evolves in functional stages (allows simultaneous development and testing).

**Observability**. "What you see is what you test."

- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.

- Internal errors are automatically reported.
- Source code is accessible.

**Controllability**. "The better we can control the software, the more the testing can be automated and optimized."

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

**Decomposability.** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

- The software system is built from independent modules.
- Software modules can be tested independently.

**Simplicity**. "The less there is to test, the more quickly we can test it."

• Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements).

• Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).

• Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

**Stability.** "The fewer the changes, the fewer the disruptions to testing."

• Changes to the software are infrequent.

• Changes to the software are controlled.

• Changes to the software do not invalidate existing tests.

• The software recovers well from failures.

**Understandability**. "The more information we have, the smarter we will test."

• The design is well understood.

• Dependencies between internal, external, and shared components are well understood.

• Changes to the design are communicated.

• Technical documentation is instantly accessible.

• Technical documentation is well organized.

• Technical documentation is specific and detailed.

• Technical documentation is accurate.

Kaner, Falk, and Nguyen suggest the following **attributes of a "good" test**:

1. **A good test has a high probability of finding an error.**

- To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position.
- A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

**2. A good test is not redundant.**

- Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.

**3. A good test should be "best of breed".**

- In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests.
- In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

**4. A good test should be neither too simple nor too complex.**

- Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors.
- In general, each test should be executed separately.

## A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process.

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

provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy

occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

**Characteristics:**

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

## TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

### VALIDATION TESTING

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between different software categories disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by

the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a Software Requirements Specification has been developed, it describes all user-visible attributes of the software and contains a Validation Criteria section that forms the basis for a validation-testing approach.

**i) Validation-Test Criteria**

Software validation is achieved through a series of tests that demonstrate conformity with requirements. 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). If a deviation from specification is uncovered, a deficiency list is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

**ii) Configuration Review**

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

**ii) Alpha and Beta Testing**

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be used; output that seemed clear to the tester may be unintelligible to a user in the field. When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time. If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software Like all other testing steps, validation tries

to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

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

**The beta test** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base. 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 from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

**SYSTEM TESTING**

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger pointing does occur, and (4) participate

in planning and design of system tests to ensure that software is adequately tested.

**i) Recovery Testing**

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur. *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 is automatic (performed by the system itself), reinitialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

**ii) Security Testing**

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain. *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack." Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more.

**WHITE-BOX TESTING**

White-box testing, called **glass-box testing** is a test case design method that uses the control structure of the procedural design to derive test cases.

Using white-box testing methods, the software engineer 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.

"Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?" or "Why don't we spend all of our energy on black-box tests?"

The answer is :

**Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.** Errors tend to creep into our work when we design and implement function, conditions, or controls that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.

**We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.** The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.

**Typographical errors are random.** When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

Each of these reasons provides an argument for conducting white-box tests. Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White-box testing is far more likely to uncover them.

## BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by **Tom McCabe** in 1976.

The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

**Flow Graph Notation**

The flow graph depicts logical control flow using the notation illustrated in Fig 5.1.

**Flow graph notation**

Each structured construct has a corresponding flow graph symbol. To illustrate the use of a flow graph, we consider the procedural design representation in Fig 5.2A. Here, a flowchart is used to depict program control structure.

**Flowchart, (A) and flow graph (B)**

- Fig maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).

- Referring to Fig, each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.4
- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.
- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
- Referring to Fig 5.3, the PDL segment translates into the flow graph shown.
- **Note:** A separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

**Fig 5.3 Compound logic**

**Cyclomatic Complexity**

**Cyclomatic complexity** is **software metric that provides a quantitative measure of the logical complexity of a program.**

Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric.

Cyclomatic complexity is defined by the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

**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.

For example, a set of independent paths for the flow graph illustrated in Fig 5.2B is

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11 path 4: 1-2-3-6-7-9-10-1-11

**Note: Each new path introduces a new edge**.

The path **1-2-3-4-5-10-1-2-3-6-8-9-10-1-11** is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Fig 5.2B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.

**Note:** The basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

**How do we know how many paths to look for?** The computation of cyclomatic complexity provides the answer.

Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P +1 where P is the number of predicate nodes contained in the flow graph G.
   The Cyclomatic complexity of the flow graph in Fig 5.2B, can be computed using each of the algorithms just noted:

   1. The flow graph has four regions.

   2. V(G) = 11 edges - 9 nodes + 2 = 4.

   3. V(G) = 3 predicate nodes + 1 = 4.

   Therefore, the cyclomatic complexity of the flow graph in Figure 17.2B is 4.

**Important**: the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

## CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing.

Other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

### Condition Testing

**Condition testing** is a test case design method that exercises the logical conditions contained in a program module.

A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator.

A **relational expression** takes the form **E1 <relational-operator> E2** where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.

A **compound condition** is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬).

A condition without relational expressions is referred to as a **Boolean expression**. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program.

Condition testing strategies have two advantages.

1. Measurement of test coverage of a condition is simple.
2. Test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

A number of condition testing strategies have been proposed.

**Branch testing** is probably the simplest condition testing strategy. For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.

**Domain testing** requires three or four tests to be derived for a relational expression. For a relational expression of the form **E1 <relational-operator> E2** three tests are required to make the value of E1 greater than, equal to, or less than that of E2. If <relational-operator> is incorrect and E1 and E2 are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in E1 and E2, a test that makes the value of E1 greater or less than that of E2 should make the difference between these two values as small as possible.

**Data Flow Testing**

The 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.

For a statement with S as its statement number,

DEF(S) = {X | statement S contains a definition of X}

USE(S) = {X | statement S contains a use of X}

If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements. To illustrate this, consider the application of DU testing to select test paths for the PDL that follows:

```
proc x
        B1;
        do while C1
        if C2
        then
        if C4
        then B4;
        else B5;
```

```
        endif;
        else
        if C3
        then B2;
        else B3;
        endif;
        endif;
        enddo;
        B6;
end proc;
```

To apply the DU testing strategy to select test paths of the control flow diagram, we need to know the definitions and uses of variables in each condition or block in the PDL.

Assume that variable X is defined in the last statement of blocks B1, B2, B3, B4, and B5 and is used in the first statement of blocks B2, B3, B4, B5, and B6. The DU testing strategy requires an execution of the shortest path from each of Bi, $0 < i \leq 5$, to each of Bj, $1 < j \leq 6$. Although there are 25 DU chains of variable X, we need only five paths to cover these DU chains. The reason is that five paths are needed to cover the DU chain of X from Bi, $0 < i \leq 5$, to B6 and other DU chains can be covered by making these five paths contain iterations of the loop.

Since the statements in a program are related to each other according to the definitions and uses of variables, the data flow testing approach is effective for error detection.

However, the problems of measuring test coverage and selecting test paths for data flow testing are more difficult than the corresponding problems for condition testing.

**Loop Testing**

Loops are the cornerstone for the vast majority of all algorithms implemented in software.

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 (Fig).

**imple 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.

4. m passes through the loop where m < n.

5. n -1, n, n + 1 passes through the loop.

**Nested loops**.

If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.

Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.

**Concatenated loops.**

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

**Unstructured loops**.

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

**Fig Classes of loops**

## BLACK-BOX TESTING

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.

That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques.

Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories:

1. Incorrect or missing
functions 2. Interface errors
3. Errors in data structures or external data base
access, 4. Behavior or performance errors, and
5. Initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.

Black-box testing purposely disregards control structure, attention is focused on the information domain.

Tests are designed to answer the following questions:

• How is functional validity tested?

- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

Black-box techniques, we derive a set of test cases that satisfy the following criteria:

1. test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing and
2. test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

**Graph-Based Testing Methods**

The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects.

Next step is to define a series of tests that verify "all objects have the expected relationship to one another".

To accomplish these steps, the software engineer begins 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.

**Fig (A) Graph notation (B) Simple example**

The symbolic representation of a graph is shown in Fig A.

Nodes are represented as circles connected by links that take a number of different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction.

A bidirectional link, called a **symmetric link**, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.

Eg. consider a portion of a graph for a word-processing application (Fig B) where

Object #1 = new file menu select

Object #2 = document window

Object #3 = document text

Referring to the figure, a menu select on new file generates a document window.

The node weight of document window provides a list of the window attributes that are to be expected when the window is generated.

The link weight indicates that the window must be generated in less than 1.0 second.

An undirected link establishes a symmetric relationship between the new file menu select and document text, and parallel links indicate relationships between document window and document text.

In reality, a far more detailed graph would have to be generated as a precursor to test case design.

The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

Beizer describes a number of behavioral testing methods that can make use of graphs:

**Transaction flow modeling**. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., flight. information. input is followed by validation/availability. processing).

**Finite state modeling**. The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., order-information is verified during inventory-availability look-up and is followed by customer-billing-information input). The state transition diagram can be used to assist in creating graphs of this type.

**Data flow modelin**g. The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node FICA.tax.withheld (FTW) is computed from gross.wages (GW) using the relationship, FTW = 0.62 - GW.

**Timing modeling**. The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.

Once nodes have been identified, links and link weights should be established.

In general, links should be named, although links that represent control flow between program objects need not be named.

Each relationship is studied separately so that test cases can be derived.

The transitivity of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, X, Y, and Z. Consider the following relationships:

X is required to compute Y

Y is required to compute Z

Therefore, a transitive relationship has been established between X and Z:

X is required to compute Z

Based on this transitive relationship, tests to find errors in the calculation of Z must consider a variety of values for both X and Y.

The symmetry of a relationship (graph link) is also an important guide to the design of test cases.

As test case design begins, the first objective is to achieve node coverage. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, link coverage is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present.

A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked

**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.

An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed.

Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

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.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, one valid and one invalid class are defined.

Example, consider data maintained as part of an automated banking application.

The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form

area code—blank or three-digit number

prefix—three-digit number not beginning with 0 or 1

suffix—four-digit number

password—six digit alphanumeric string

commands—check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking application can be specified as

**area code:** Input condition, Boolean—the area code may or may not be present.

Input condition, range—values defined between 200 and 999, with specific exceptions.

**prefix:** Input condition, range—specified value >200

Input condition, value—four-digit length

**password:** Input condition, Boolean—a password may or may not be present.

Input condition, value—six-character string.

**command:** Input condition, set—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

**Boundary Value Analysis**

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.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers.

3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.
   By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

**Comparison Testing**

There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical.

In such applications redundant hardware and software are often used to minimize the possibility of error.

When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification.

In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

Researchers have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system.

These independent versions form the basis of a black-box testing technique called **comparison testing** or **back-to-back testing**.

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software.

If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference. In most cases, the comparison of outputs can be performed by an automated tool.

Comparison testing is not foolproof. If the specification from which all versions have been developed is in error, all versions will likely reflect the error. In addition, if each of the

independent versions produces identical but incorrect results, condition testing will fail to detect the error.

**A geometric view of test cases**

**Orthogonal Array Testing**

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small, it is possible to consider every input permutation and exhaustively test processing of the input domain.

However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing become impractical or impossible.

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 errors associated with region faults—an error category associated with faulty logic within a software component.

**SUBJECT NAME : SOFTWARE ENGINEERING** **CLASS : II B.Sc. (CT)**
**SUBJECT CODE : 17CTU402** **UNIT V** **SEMESTER : IV**
**BATCH (2017-2020)**

## PART A(Online)

## PART B (2 Marks)

1. Define abstraction.

2. What do you mean by an error?

3. Differentiate between refinement and refactoring

4. Compare black box and white box testing

5. Write the difference between transform flow and transaction flow

6. List the different types of loops in testing

7. What is transform mapping?

8. What is validation testing?

9. Define transaction mapping.

10. What is the use of system testing?

## PART C (6 Marks)

1. Explain Graph based testing methods in Black Box testing.

2. Demonstrate Flow graph notation and Independent program path in Basis path testing.

3. Demonstrate in detail about Validation testing

4. Explain in detail about Equivalence Partitioning

5. Discuss about Boundary value analysis.

6. Write in detail about Software Testing Fundamentals.

7. Illustrate in detail about System testing.

8. Illustrate the use of dataflow testing in software engineering process.

9. Discuss in detail about orthogonal array testing.

10. Illustrate loop testing and its types.

| | **Software Enginering (17CTU402)** | | **UNIT V** | |
|---|---|---|---|---|
| **S.No** | **Question** | **Option A** | **Option B** | **Option C** |
| 1 | _____ is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. | software specification | software generation | software coding |
| 2 | Software is tested from _____ different perspectives. | 2 | 3 | 4 |
| 3 | Software engineers are by their nature _____ people. | pessimistic | optimistic | constructive |
| 4 | _____ is a process of executing a program with the intent of finding an error. | coding | testing | debugging |
| 5 | All tests should be _____ to customer requirements. | traceable | designed | tested |
| 6 | Tests should be planned long before _____ begins. | testing | coding | specification |
| 7 | Testing should begin in the _____ and progress toward testing in the large. | design | beginning | small |
| 8 | The less there is to test, the more _____ we can test it. | quickly | shortly | automatically |
| 9 | _____ is a process of executing a program with the intend of finding an error. | testing | coding | planning |

| | | | | |
|---|---|---|---|---|
| 10 | A good _____ is one that has a high probability of finding an as-yet-undiscovered error | planning | test case | objective |
| 11 | All _____ should be traceable to customer-requirements. | analysis | designs | tests |
| 12 | _____ is simple how easily a computer program can be tested. | software operability | software simplicity | software decomposability |
| 13 | The better it works, the more efficiently it can be testing. This characteristic is called _____. | operability | observability | controllability |
| 14 | There are _____ characteristics in testability | 5 | 6 | 7 |
| 15 | What you see is what you test. This characteristic is called _____. | controllability | observability | decomposability |
| 16 | The better we can control the software, the more the testing can be automated and optimized. This characteristic is called _____. | operability | stability | understandability |
| 17 | By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting. This characteristic is called _____. | decomposability | simplicity | stability |
| 18 | . The less there is to test, the more quickly we can test it. This characteristic is called _____. | controllability | simplicity | operability |
| 19 | The fewer the changes, the fewer the disruptions to testing. This characteristic is called _____. | controllability | decomposability | stability |

| | | | | |
|---|---|---|---|---|
| 20 | . The more information we have, the smarter we will test. This characteristic is called _____. | controllability | decomposability | stability |
| 21 | A good test has a high _____ of finding an error. | probability | simplicity | understandability |
| 22 | A good test is not _____. | stable | redundant | simple |
| 23 | White-box testing sometimes called _____. | control structure testing | condition testing | glass-box testing |
| 24 | Logic errors and incorrect assumptions are inversely proportional to the _____ that a program path will be executed | simplicity | probability | understandability |
| 25 | Typographical errors are _____. | redundant | simple | random |
| 26 | One often believes that a _____ path is not likely to be executed when, in fact, it may be executed on a regular basis. | control | structural | physical |
| 27 | Basic path testing is a _____. | black-box testing | white-box testing | control structure testing |
| 28 | _____ is a software metric that provides a quantitative measure of the logical complexity of a program. | cyclomatic complexity | flow graph | deriving test cases |
| 29 | An _____ is any path through the program that introduces atleast one new set of processing statements or a new condition. | dependent path | independent path | basic path |
| 30 | There are _____ steps to be applied to derive the basis set. | 2 | 3 | 4 |
| 31 | There are _____ test cases that satisfy the basis set. | 3 | 4 | 5 |

| | | | | |
|---|---|---|---|---|
| 32 | . A _____ is a square matrix whose size is equal to the number of nodes on the flow graph. | graph matrix | matrix | flow graph |
| 33 | To develop a software tool that assists in basis path testing, a data structure called a _____ is useful. | matrix | flow graph | graph matrix |
| 34 | _____ requires three or four tests to be derived for a relational expression. | branch testing | data flow testing | data control testing |
| 35 | _____ is probably the simplest condition testing strategy. | branch testing | data flow testing | condition testing |
| 36 | The _____ method selects test paths of a program according to the locations of definitions and uses of variables in the program | data flow testing | condition testing | loop testing |
| 37 | _____ is a white box testing technique that focuses exclusively on the validity of loop constructions | data flow testing | loop testing | condition testing |
| 38 | _____ is a test case design method that exercises the logical conditions contained in a program module | black box testing | loop testing | data flow testing |
| 39 | _____ is called behavioral testing. | black box testing | loop testing | data flow testing |
| 40 | The first step in _____ is to understand the objects that are modeled in software and the relationships that connect these objects | black box testing | loop testing | data flow testing |

| | | | | |
|---|---|---|---|---|
| 41 | Equivalence partitioning is a _____ method that divides the input domain of a program into classes of data. | black box testing | loop testing | data flow testing |
| 42 | Comparison testing is also called _____. | black box testing | loop testing | behavioral testing |
| 43 | _____ testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. | orthogonal array | loop | behavioral |
| 44 | _____ focuses verification effort on the smallest unit of software design – the software component or module. | module testing | unit testing | structure testing |
| 45 | A driver is nothing more than a _____. | subprogram | main program | stub |
| 46 | _____ serve to replace modules that are subordinate called by the component to be tested. | subprograms | main programs | stubs |
| 47 | Drivers and _____ represent overhead. | subprograms | main programs | stubs |
| 48 | _____ of execution paths is an essential task during the unit test. | unit testing | module testing | selective testing |
| 49 | Good _____ dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does occur | design | testing | code |

| | | | | |
|---|---|---|---|---|
| 50 | _____ is completely assembled as a package, interfacing errors have been uncovered and corrected. | software | program | code |
| 51 | The Process of Configuration identification involves the specification of components in the software project are known as _____. | Configuration Items | Change Control | Configuration Control |
| 52 | Implementing a quality system is spent on writing documents which specify how certain tasks are to be carried out is known as _____. | Procedures | Policies | Function |
| 53 | _____ task involves the programmer to receive a specification of a module. | Integration Programming | System Programming | Unit Testing |
| 54 | Quality Assurance follows _____ methodology | Defect analysis | Defect Prevention | Error detection |
| 55 | Quality Assurance is based on _____ work | Product Oriented | Function Oriented | Process Oriented |
| 56 | SEI means | Software Engineering Institute | Software Engineering International | Software Engineering Independent |
| 57 | ISO means | Internal Organizations for standards | Intermediate Organizations for standards | International Organizations for standards |
| 58 | PCMM means | Personal Capability Maturity Model | People Capability Maturity Model | Professional Capability Maturity Model |

| | | | | |
|---|---|---|---|---|
| 59 | Quality Control is based on _____ methodology | Defect Prevention | Process Oriented | Defect Detection |
| 60 | the document shows the relationship between requirement specification and test case is called _____ | Matrix | Traceability Matrix | Defect Analysis |

| | | | |
|---|---|---|---|
| **Option D** | | | **Answer** |
| software testing | | | software testing |
| 5 | | | 2 |
| destructive | | | constructive |
| designing | | | testing |
| coded | | | traceable |
| requirements | | | testing |
| big | | | small |
| hardly | | | quickly |
| designing | | | testing |

| | | | |
|---|---|---|---|
| goal | | | test case |
| plans | | | tests |
| software testability | | | software testability |
| decomposability | | | operability |
| 8 | | | 7 |
| stability | | | observability |
| controllability | | | controllability |
| understandability | | | decomposability |
| observability | | | simplicity |
| understandability | | | stability |

| | | | |
|---|---|---|---|
| understandability | | | understandability |
| stability | | | probability |
| complex | | | redundant |
| black-box testing | | | glass-box testing |
| stability | | | probability |
| complex | | | random |
| logical | | | logical |
| control path testing | | | white-box testing |
| graph matrices | | | cyclomatic complexity |
| control path | | | independent path |
| 5 | | | 4 |
| 6 | | | 6 |

| | | | |
|---|---|---|---|
| cyclomatic complexity | | | graph matrix |
| cyclomatic omplexity | | | graph matrix |
| domain testing | | | domain testing |
| domain testing | | | branch testing |
| black box testing | | | data flow testing |
| control path testing | | | loop testing |
| condition testing | | | condition testing |
| condition testing | | | black box testing |
| condition testing | | | black box testing |

| | | | |
|---|---|---|---|
| condition testing | | | black box testing |
| back-to-back testing | | | back-to-back testing |
| back-to-back | | | orthogonal array |
| system testing | | | unit testing |
| subroutine | | | main program |
| subroutines | | | stubs |
| subroutines | | | stubs |
| white box testing | | | selective testing |
| module | | | design |

| | | | |
|---|---|---|---|
| all of the above | | | software |
| Project control | | | Configuration Items |
| Definitions | | | Procedures |
| Configuration Control | | | Unit Testing |
| Error correction | | | Defect Prevention |
| Design Oriented | | | Process Oriented |
| System Engineering Institute | | | Software Engineering Institute |
| Internal optimization standards | | | International Organizations for standards |
| Project Capability Maturity Model | | | People Capability Maturity Model |

| | | | |
|---|---|---|---|
| debugging | | | Defect Detection |
| Matrix Analysis | | | Traceability Matrix |

**Reg.No** _____

**[17CTU402]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University)**
**(Established Under Section 3 of UGC Act 1956)**
**Coimbatore – 641 021**

**COMPUTER TECHNOLOGY**

**Fourth Semester**
**FIRST INTERNAL EXAMINATION – DECEMBER 2018**

**SOFTWARE ENGINEERING**

**Class: III B.Sc. IT (A&B)** **Duration :** 2 Hours
**Date & Session : .12.2018** **Maximum :** 50 Marks

---

**PART-A (20 X 1 = 20 Marks)**
**Answer ALL the Questions**

1.Software takes on a _____ role.
a) single b) **dual** c) triple d) tetra

2. Software is a _____.
a) virtual b) system c) **modifier** d) framework

3. Software doesn't _____.
a) tearout b) **wearout** c) degrade d) deteriorate

4. Instructions that when executed provide desired function and performance is called____.
a) **software** b) hardware c) firmware d) humanware

5. Software will undergo _____.
a) database b) testing c) **enhancement** d) manufacture

6. The first step to develop software is _____.
a) design b) **requirements gathering** c) coding d) analysis

7. Software engineering activities include _____.
**a)** decision b) affliction c) hardware d) **maintenance**

8. All process model prescribes a _____.
a) circular b) elliptical c) spiral d) **workflow**

9. Component based development incorporates the characteristics of the _____ model.
a) **spiral** b) hierarchical c) circular d) elliptical

10. Prototype is a _____.
a) software b) hardware c) computer d) **model**

11. Software project management begins with a set of activities that are called _____.
a) **project planning** b) software scope c) software estimation d) decomposition

12. Breaking up of a complex problem into small steps is called _____.
**a)** project planning b) software scope c) software estimation d) **decomposition**

13. The ease with which software can be transferred from one computer to another. This quality attribute is called _____.
a) **portability** b) reliability c) efficiency d) accuracy

14. The ability of a program to perform a required function under stated condition for a stated period of time. This quality attribute is called _____.
a) **reliability** b) accuracy c) portability d) efficiency
15. The extent software can continue to operate correctly. This quality attribute is called _____.
a) **robustness** b) correctness c) efficiency d) reliability
16. The bedrock that supports software engineering is a_____.
a) tools b) methods c) process models d) **quality focus**
17. Software requirements analysis work products must be reviewed for _____.
a) modeling b) **completeness** c) information processing d) functional requirement
18. Software requirements analysis is divided into _____ areas of effort.
a) 2 b) 3 c) **4** d) 5
19. Entity is a _____.
a) data b) information c) model d) **physical thing**
20. Transformations are represented by _____.
a) labeled arrows b) **bubbles** c) entity d) label

**PART-B (3 X 2 = 6 Marks)**
**(Answer ALL the Questions)**

21. Define software characteristics.

**Software Characteristics**

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

22. Define Quality focus.

Any engineering approach much rests on organizational approach to quality, e.g. total quality management and such emphasize continuous process improvement (that is increasingly more effective approaches to software engineering). The bedrock that supports a software engineering is a ***quality focus***.

23. What is Data flow model.

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

**Guidelines**

1. The level 0 data flow diagram should depict the software/system as a single bubble
2. Primary input and output should be carefully noted
3. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level
4. All arrows and bubbles should be labeled with meaningful names
5. Information flow continuity must be maintained from level to level
6. One bubble at a time should be refined.

**PART-C (3 X 8 = 24 Marks)**
**(Answer ALL the Questions)**

24. a) Describe about evolving role of Software.

**The Evolving Role of Software**

Software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer— producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information.

Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance

competitiveness; it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

**Software**

**Computer software**, or just **software**, is a collection of computer programs and related data that provide the instructions for telling a computer what to do and how to do it. In other words, software is a conceptual entity which is a set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system. We can also say software refers to one or more computer programs and data held in the storage of the computer for some purposes.

In other words software is a set of **programs, procedures, algorithms** and its **documentation**. Program software performs the function of the program it implements, either by directly providing instructions to the computer hardware or by serving as input to another piece of software.

The term was coined to contrast to the old term hardware (meaning physical devices). In contrast to hardware, software is intangible, meaning it "cannot be touched". Software is also sometimes used in a more narrow sense, meaning application software only. Sometimes the term includes data that has not traditionally been associated with computers, such as film, tapes, and records.

(or)

b) Explain about Software Process Framework activities.

**Process Framework**

Identifies a small number of framework activities that are applicable to all software projects. In addition the framework encompasses umbrella activities that are

applicable across the software process.

**Generic Process Framework Activities**

Each framework activity is populated by a set of software engineering actions. An action, e.g. design, is a collection of related tasks that produce a major software engineering work product.

**Communication** – lots of communication and collaboration with customer and other stakeholders.. Encompasses requirements gathering.

**Planning** – establishes plan for software engineering work that follows. Describes technical tasks, likely risks, required resources, works products and a work schedule

**Modeling** – encompasses creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements.

**Modeling Activity – composed of two software engineering actions**

• **analysis** – composed of work tasks (e.g. requirement gathering, elaboration, specification and validation) that lead to creation of analysis model and/or requirements specification.

• **design** – encompasses work tasks such as data design, architectural design, interface design and component level design leads to creation of design model and/or a design specification.

**Construction** – code generation and testing.

**Deployment** – software, partial or complete, is delivered to the customer who evaluates it and provides feedback.

Different projects demand different task sets. Software team chooses task set based on problem and project characteristics.

25. a) Discuss about the Layered Technology of Software Engineering.

**Software Engineering as a Layered Technology**

Any engineering approach much rests on organizational approach to quality, e.g. total quality management and such emphasize continuous process improvement (that is increasingly more effective approaches to software engineering). The bedrock that supports a software engineering is a ***quality focus***.

The foundation for software engineering is the ***process*** layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of *key process areas* (KPAs) that must be established for effective delivery of

software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.


**Fig 1.3 Software Engineering Layers**

Software engineering ***methods*** provide the technical how-to's for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

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

(or)

b) Explain about RAD Process model.

**1. The RAD Model**

*Rapid application development* (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaptation of the waterfall model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days)

Like other process models, the RAD approach maps into the generic framework activities.

- Communication works to understand the business problem and the information characteristics that the software must accommodate.
- Planning is essential because multiple software teams work in parallel on different system functions.
- Modeling encompasses three major phases- business modeling, data modeling and process modeling and establishes design representations that serve as the basis for RAD's construction activity.
- Construction emphasizes the use of preexisting software components and the application of automatic code generation.
- Finally, deployment establishes a basis for subsequent iterations, if required.

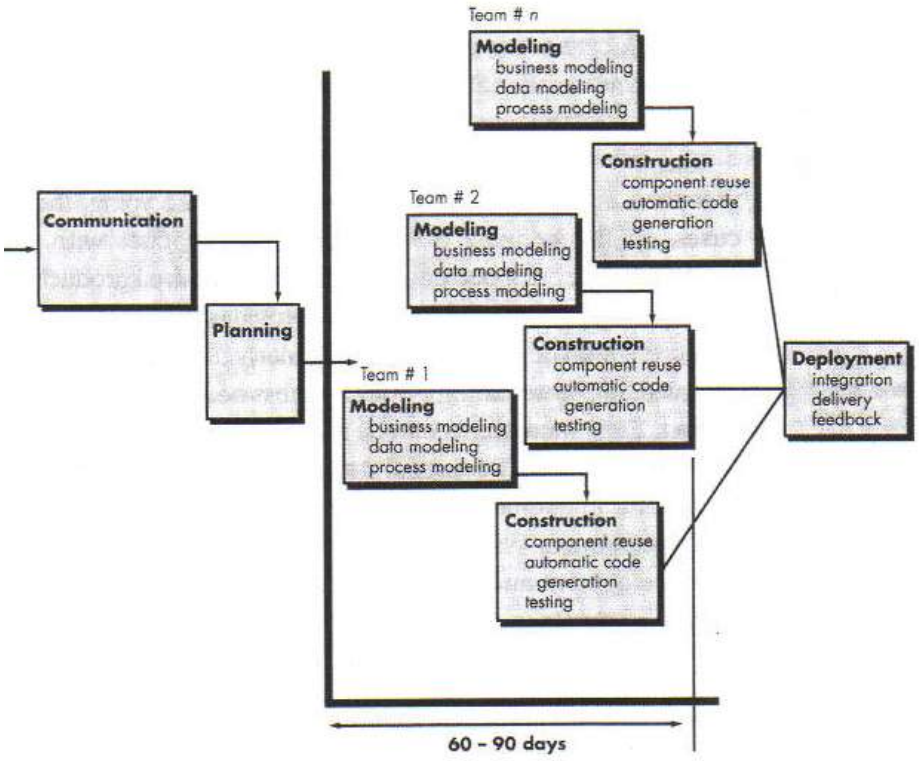
The RAD process model is illustrated in Fig 1.6. Obviously, the time constraints imposed on a RAD project demand "scalable scope" . If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

**Drawbacks**

• For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

• If developers and customers are not committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame, RAD projects will fail.

**Fig:1.6 The RAD Model**



- If a system cannot be properly modularized, building the components necessary for RAD will be problematic
- If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
- RAD may not be appropriate when technical risks are high.

26. a) Illustrate about the techniques of Requirement Analysis.

**Requirement Analysis**

Requirement Analysis results in the specification of software's operational characteristics indicates software interface with other system elements and establishes constraints that software must meet

Requirement analysis allow the software engineer to elaborate on basic requirements established during earlier requirement engineering tasks and build models that depict user scenario, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

Requirement analysis provides the software designer with a representation of information, function and behavior that can be translated to architectural, interface and component-level designs

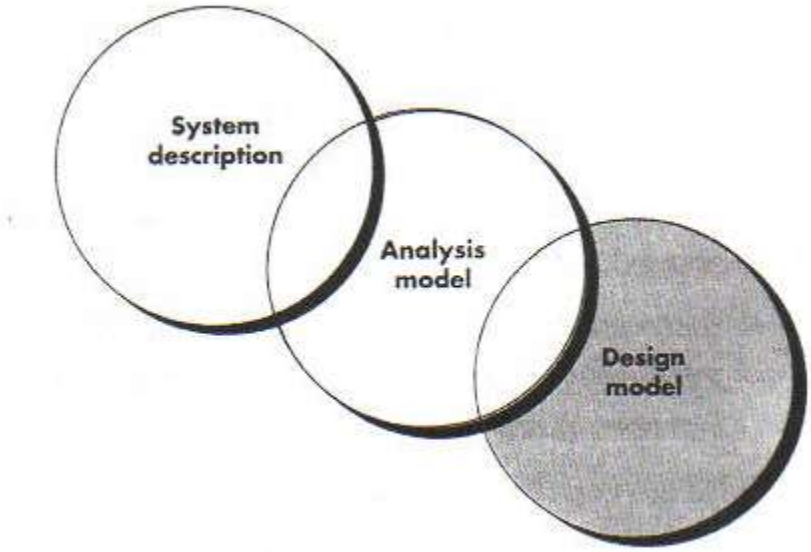
Finally, the analysis model and the requirement specification provide the developer and customer with the means to assess quality once software is built. Throughout analysis modeling, the software engineer's primary focus is on what and not how

**1. Overall Objectives and Philosophy**

The analysis model must have three primary objectives

- To describe what the customer requires
- To establish a basis for the creation of software design
- To define a set of requirements that can be validated once the software is built

The analysis model bridges the gap between a system level description that describes overall system functionality as it is achieved by applying software, hardware, data, human and other system elements and a software design that describes the software's application architecture, user interface and component level structure

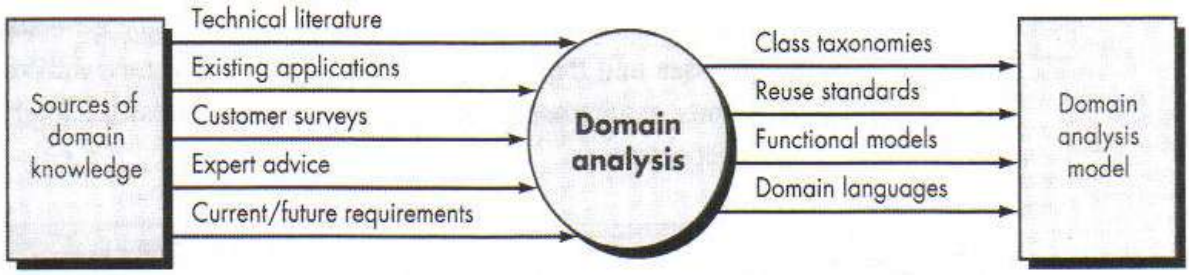



**2. Analysis rules of Thumb**

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system
- Delay consideration of infrastructure and non functional models until design
- Minimize coupling throughout the system
- Be certain that the analysis model provides value to all stakeholders
- Keep the model as simple as it can be

**3. Domain Analysis**

The analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows a software engineer or analyst to recognize and reuse them, the creation of the analysis model is expedited.



Input and output of Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within the application domain.

(or)

b)Explain the concept of Data Modeling and Data Objects.

**Data Modeling Concepts**

Analysis modeling often begins with data modeling. The software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.

**1. Data object**

A *data object* is a representation of almost any composite information that must be understood by software. By *composite information,* we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car (Figure 12.2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

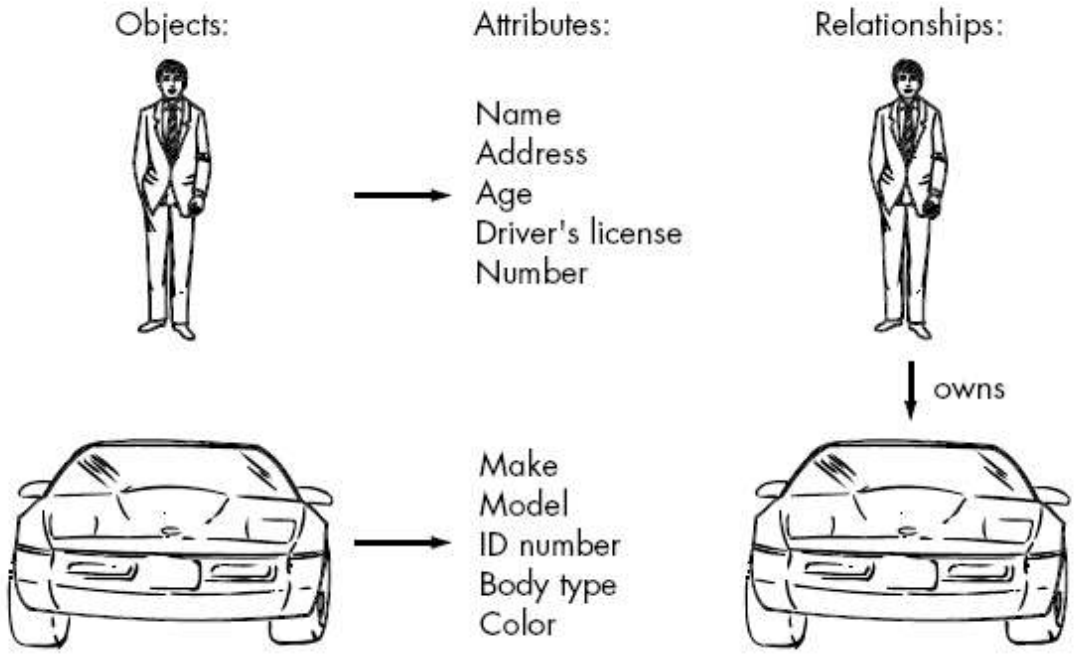
**2. Data Attributes**

Attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

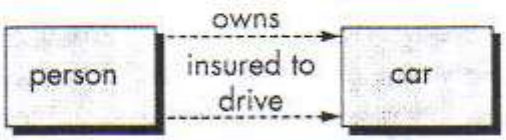
(3) make reference to another instance in another table.




**3. Relationships**

Data objects are connected to one another in different ways. Consider two data objects, person and car. These objects can be represented using the simple notation illustrated in below Figure. A connection is established between person and car because the two objects are related.


(a) A basic connection between data objects




(b) Relationships between data objects

**4. Cardinality and Modality**

The elements of data modeling—data objects, attributes, and relationships— provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that

bind them. But a simple pair that states: **object X** *relates* to **object Y** does not provide enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called *cardinality.*

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

Cardinality defines "the maximum number of objects that can participate in a relationship"

**Modality**

The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

Reg.No _____

[17CTU402]

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University)**
**(Established Under Section 3 of UGC Act 1956)**
**Coimbatore – 641 021**

**COMPUTER TECHNOLOGY**
**Fourth Semester**
**SECOND INTERNAL EXAMINATION – FEBRUARY 2019**

**SOFTWARE ENGINEERING**

**Class: II B.Sc. CT** **Duration :** 2 Hours
**Date & Session : 04.02.2019 & AN** **Maximum :** 50 Marks

---

**PART-A (20 X 1 = 20 Marks)**
**Answer All the Questions**

1. The _____ becomes the foundation for design, providing the designer with an essential representation of software that can be mapped in to an implementation context.
   a) prototype b) model c) interface d) software
2. The _____ is one method for representing the behavior of a system by depicting its state and events.
   a) state diagram b) use case diagram c) ER diagram d) DFD
3. When an sensor event is recognized, the _____ invokes an audible alarm attached to the system.
   a) model b) software c) delay d) prototype
4. A _____ is always a model – an abstraction of some real situation that is normally quite complex.
   a) software b) prototype d) specification d) function
5. Design is not _____, coding is not design.
   a) coding b) analysis c) review d) event
6. The design _____ is the equivalent of an architect's plan for a house.
   a) analysis b) process c) model d) function
7. At the highest level of _____, a solution is stated in broad terms, using the language of the problem environment.
   a) refinement b) modularity c) abstraction d) continuity
8. We should allow user interaction to be _____ and undoable.
   a) interruptible b) flexible c) rigid d) encouraging
9. We should allow user interaction to interruptible and _____.
   a) undoable b) flexible c) rigid d) encouraging
10. Design begins with the _____ model.
    a) data b) requirements c) specification d) code
11. Software design methodologies lack the _____ that are normally associated with more classical engineering design disciplines.
    a) depth b) flexibility c) quantitative nature d) all of the above

12. We should disclose information in a _____ fashion.
    a) open b) progressive c) streamline d) flexible
13. The visual layout of the _____ should be based on a real world metaphor.
    a) interaction modes b) interface c) design d) structure
14. The interface should present and acquire _____ in a consistent fashion.
    a) information b) task c) knowledge d) idea
15. The _____ is a primary concern of software engineers.
    a) software design b) software maintenance c) software product d) software quality
16. The quality attributes for very software product includes _____.
    a) design b) clarity c) accuracy d) visibility
17. The software requirements specification is developed as a consequence of _____.
    a) review b) analysis c) prototyping d) functional description
18. The designer's goal is to produce a model or representation of_____ that will later be built.
    a) component b) entity c) data d) raw material
19. The information domain contains _____ different views of the data and control as each is processed by a computer program.
    a) 2 b) 3 c) 4 d) 5
20. The content of _____ is defined by the attributes that are needed to create it.
    a) system status b) functional model c) paycheck d) behavioral model

**PART-B (3 X 2 = 6 Marks)**
**(Answer All the Questions)**

21. Define Data attributes.
22. What is Project Risk?
23. What is Quality Management?

**PART-C (3 X 8 = 24 Marks)**
**(Answer All the Questions)**

24. a) Explain about the characteristics and components of SRS.
    (or)
    b) Describe about Data Flow model.

25. a) Explain about the concept of Software Risk.
    (or)
    b) Illustrate about the Metric of Process.

26. a) Discuss about Risk Projection and Risk Refinement.
    (or)
    b) Explain about RMMM plan.

Reg.No _____

[17CTU402]

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University)**
**(Established Under Section 3 of UGC Act 1956)**
**Coimbatore – 641 021**

**COMPUTER TECHNOLOGY**
**Fourth Semester**
**SECOND INTERNAL EXAMINATION – FEBRUARY 2019**

**SOFTWARE ENGINEERING**

**Class: II B.Sc. CT** **Duration :** 2 Hours
**Date & Session : 04.02.2019 & AN** **Maximum :** 50 Marks

---

**PART-A (20 X 1 = 20 Marks)**
**Answer All the Questions**

1. The _____ becomes the foundation for design, providing the designer with an essential representation of software that can be mapped in to an implementation context.
   a) prototype b) **model** c) interface d) software
2. The _____ is one method for representing the behavior of a system by depicting its state and events.
   a) **state diagram** b) use case diagram c) ER diagram d) DFD
3. When an sensor event is recognized, the _____ invokes an audible alarm attached to the system.
   a) model b) **software** c) delay d) prototype
4. A _____ is always a model – an abstraction of some real situation that is normally quite complex.
   a) software b) prototype d) **specification** d) function
5. Design is not _____, coding is not design.
   a) **coding** b) analysis c) review d) event
6. The design _____ is the equivalent of an architect's plan for a house.
   a) analysis b) process c) **model** d) function
7. At the highest level of _____, a solution is stated in broad terms, using the language of the problem environment.
   a) refinement b) modularity c) **abstraction** d) continuity
8. We should allow user interaction to be _____ and undoable.
   a) **interruptible** b) flexible c) rigid d) encouraging
9. We should allow user interaction to interruptible and _____.
   a) **undoable** b) flexible c) rigid d) encouraging
10. Design begins with the _____ model.
    a) data b) **requirements** c) specification d) code
11. Software design methodologies lack the _____ that are normally associated with more classical engineering design disciplines.
    a) depth b) flexibility c) quantitative nature d) **all of the above**

12. We should disclose information in a _____ fashion.
    a) open b) **progressive** c) streamline d) flexible
13. The visual layout of the _____ should be based on a real world metaphor.
    a) interaction modes b) **interface** c) design d) structure
14. The interface should present and acquire _____ in a consistent fashion.
    a) **information** b) task c) knowledge d) idea
15. The _____ is a primary concern of software engineers.
    a) software design b) software maintenance c) software product d) **software quality**
16. The quality attributes for very software product includes _____.
    a) design b) **clarity** c) accuracy d) visibility
17. The software requirements specification is developed as a consequence of _____.
    a) review b) **analysis** c) prototyping d) functional description
18. The designer's goal is to produce a model or representation of_____ that will later be built.
    a) **component** b) entity c) data d) raw material
19. The information domain contains _____ different views of the data and control as each is processed by a computer program.
    a) 2 b**) 3** c) 4 d) 5
20. The content of _____ is defined by the attributes that are needed to create it.
    a) system status b) functional model c**) paycheck** d) behavioral model

**PART-B (3 X 2 = 6 Marks)**
**(Answer All the Questions)**

21. Define Data attributes.

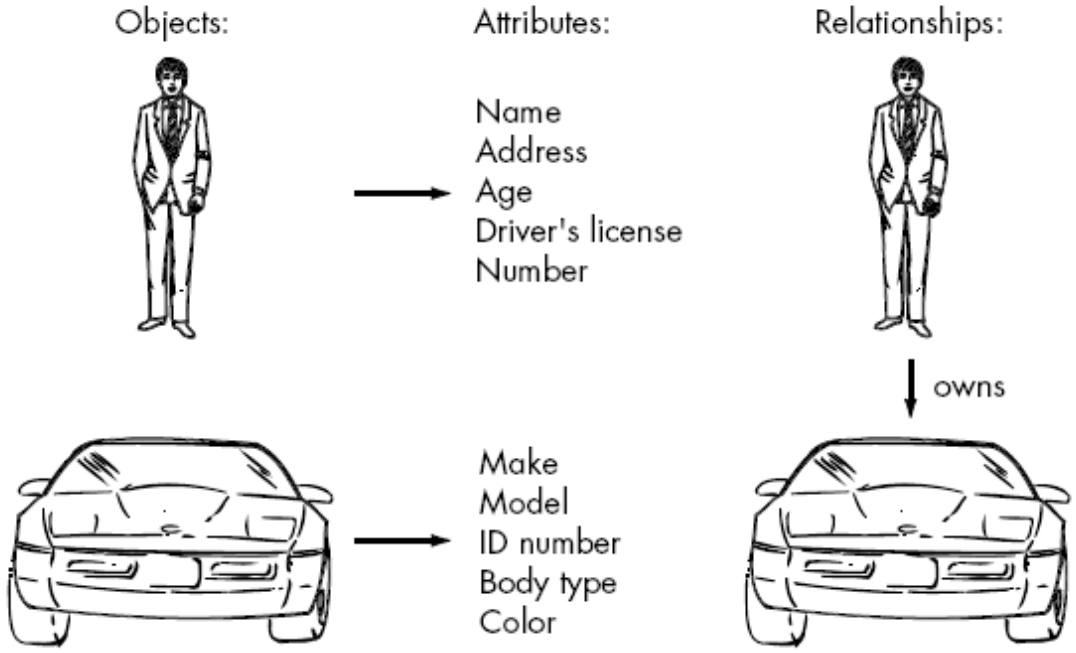
**Data Attributes**

Attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table.




22. What is Project Risk?

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

23. What is Quality Management?

**Quality Management**

The drumbeat for improved software quality began in earnest as software became increasingly integrated in every facet of our lives. By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised. Worse, both government and industry became increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more. By the turn of the century, CIO Magazine trumpeted the headline, "Let's Stop Wasting $78 Billion a Year," lamenting the fact that "American businesses spend billions for software that doesn't do what it's supposed to do". InformationWeek echoed the same concern: Despite good intentions, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of

computer-system downtime and costing U.S. companies about $100 billion last year in lost productivity and repairs, says the Standish Group, a market research firm. That doesn't include the cost of losing angry customers. In 2005, Computer World lamented that "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction. A year later, InfoWorld wrote about the "the sorry state of software quality" reporting that the quality problem had not gotten any better. developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated.

**PART-C (3 X 8 = 24 Marks)**
**(Answer All the Questions)**

24. a) Explain about the characteristics and components of SRS.

**Characteristics of an SRS**

Software requirements specification should be accurate, complete, efficient, and of high quality, so that it does not affect the entire project plan. An SRS is said to be of high quality when the developer and user easily understand the prepared document. Other characteristics of SRS are discussed below.

- Correct
- Complete
- Unambiguous
- Verifiable
- Consistent
- Ranked for importance and/or stability
- Modifiable
- Traceable

1. **Correct:** SRS is correct when all user requirements are stated in the requirements document. The stated requirements should be according to the desired system. This implies that each requirement is examined to ensure that it (SRS) represents user requirements. Note that there is no specified tool or procedure to assure the correctness of SRS. Correctness ensures that all specified requirements are performed correctly.
2. **Unambiguous:** SRS is unambiguous when every stated requirement has only one interpretation. This implies that each requirement is uniquely interpreted. In case there is a term used with multiple meanings, the requirements document should specify the meanings in the SRS so that it is clear and easy to understand.

3. **Complete:** SRS is complete when the requirements clearly define what the software is required to do. This includes all the requirements related to performance, design and functionality.
4. **Ranked for importance/stability:** All requirements are not equally important, hence each requirement is identified to make differences among other requirements. For this, it is essential to clearly identify each requirement. Stability implies the probability of changes in the requirement in future.
5. **Modifiable:** The requirements of the user can change, hence requirements document should be created in such a manner that those changes can be modified easily, consistently maintaining the structure and style of the SRS.
6. **Traceable:** SRS is traceable when the source of each requirement is clear and facilitates the reference of each requirement in future. For this, forward tracing and backward tracing are used. Forward tracing implies that each requirement should be traceable to design and code elements. Backward tracing implies defining each requirement explicitly referencing its source.
7. **Verifiable:** SRS is verifiable when the specified requirements can be verified with a cost-effective process to check whether the final software meets those requirements. The requirements are verified with the help of reviews. Note that unambiguity is essential for verifiability.
8. **Consistent:** SRS is consistent when the subsets of individual requirements defined do not conflict with each other. For example, there can be a case when different requirements can use different terms to refer to the same object. There can be logical or temporal conflicts between the specified requirements and some requirements whose logical or temporal characteristics are not satisfied. For instance, a requirement states that an event 'a' is to occur before another event 'b'. But then another set of requirements states (directly or indirectly by transitivity) that event 'b' should occur before event 'a'.

COMPONENTS OF THE SRS
Introduction to Components of the SRS

In previous section, we discussed various characteristics that will help in completely specification the requirements. Here we describe some of system properties that an SRS should specify. The basic issues, an SRS must address are:

Functional requirements

Performance requirements

Design constraints

External interface requirements

Conceptually, any SRS should have these components. Now we will discuss them one by one.

1. Functional Requirements

Functional requirements specify what output should be produced from the given inputs. So they basically describe the connectivity between the input and output of the system. For each functional requirement:

1. A detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs be specified:

2. All the operations to be performed on the input data obtain the output should be specified, and

3. Care must be taken not to specify any algorithms that are not parts of the system but that may be needed to implement the system.

4. It must clearly state what the system should do if system behaves abnormally when any invalid input is given or due to some error during computation. Specifically, it should specify the behaviour of the system for invalid inputs and invalid outputs.

2. Performance Requirements (Speed Requirements)

This part of an SRS specifies the performance constraints on the software system. All the requirements related to the performance characteristics of the system must be clearly specified. Performance requirements are typically expressed as processed transaction s per second or response time from the system for a user event or screen refresh time or a combination of these. It is a good idea to pin down performance requirements for the most used or critical transactions, user events and screens.

2. Design Constraints

The client environment may restrict the designer to include some design constraints that must be followed. The various design constraints are standard compliance, resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Standard Compliance: It specifies the requirements for the standard the system must follow. The standards may include the report format and according procedures.

Hardware Limitations: The software needs some existing or predetermined hardware to operate, thus imposing restrictions on the design. Hardware limitations can includes the types of machines to be used operating system availability memory space etc.

Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive, so they should be minimized.

Security: Currently security requirements have become essential and major for all types of systems. Security requirements place restriction s on the use of certain commands control access to database, provide different kinds of access, requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

4. External Interface Requirements

For each external interface requirements:

1. All the possible interactions of the software with people hardware and other software should be clearly specified,

2. The characteristics of each user interface of the software product should be specified and

3. The SRS should specify the logical characteristics of each interface between the software product and the hardware components for hardware interfacing.

(or)

b) Describe about Data Flow model.

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows and the transformations are represented by circles (also called bubbles). The DFD is presented in hierarchical fashion. That is, the first data flow model sometimes called a level 0 DFD or context diagram represent the system as a whole.

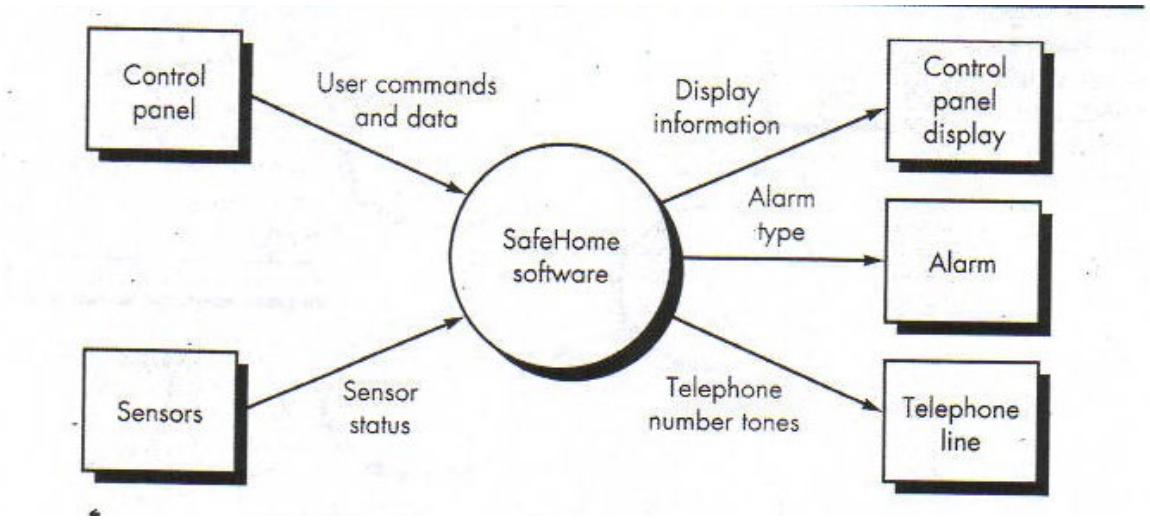
**1. Creating a data flow model**

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

**Guidelines**

1. The level 0 data flow diagram should depict the software/system as a single bubble
2. Primary input and output should be carefully noted
3. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level
4. All arrows and bubbles should be labeled with meaningful names
5. Information flow continuity must be maintained from level to level
6. One bubble at a time should be refined.

**Context level DFD for the safe home security function**



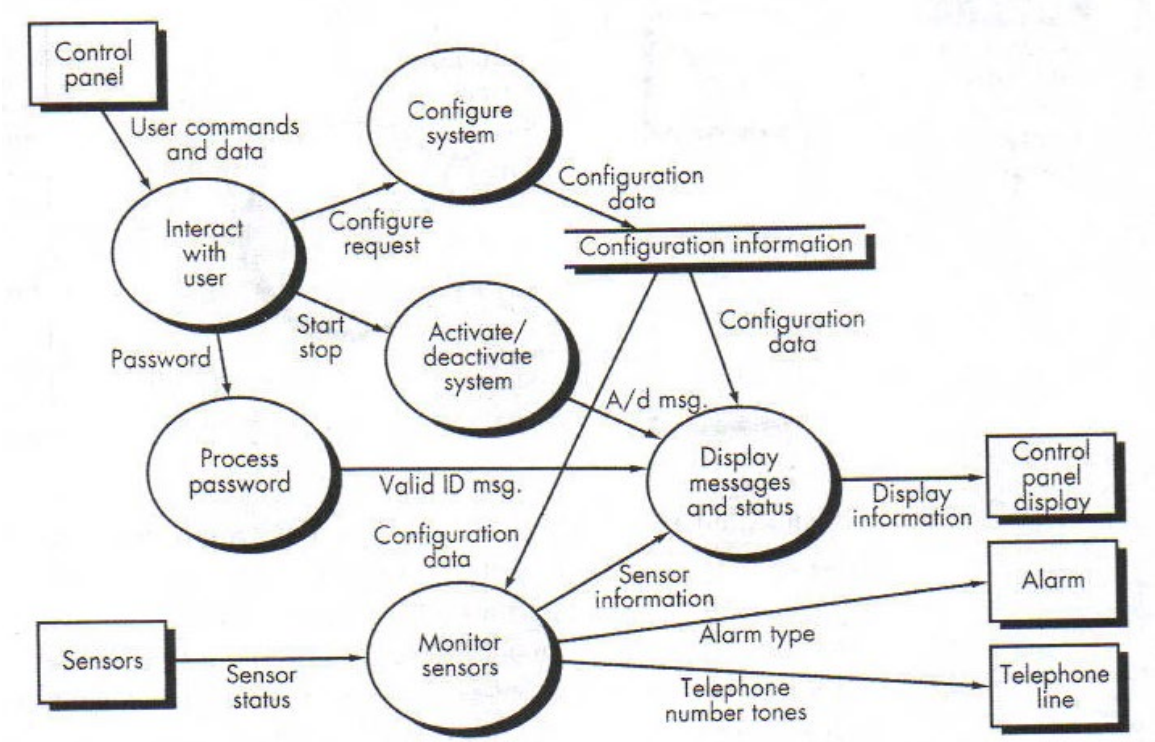
The safe home security function enables the homeowner to configure the security system. When it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through the internet, a PC, or a control panel

During installation, the safe home PC is used to program and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

When a sensor event is recognized, the software involves an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until a telephone connection is obtained

The level 0 DFD is now expanded into a level 1 data flow model

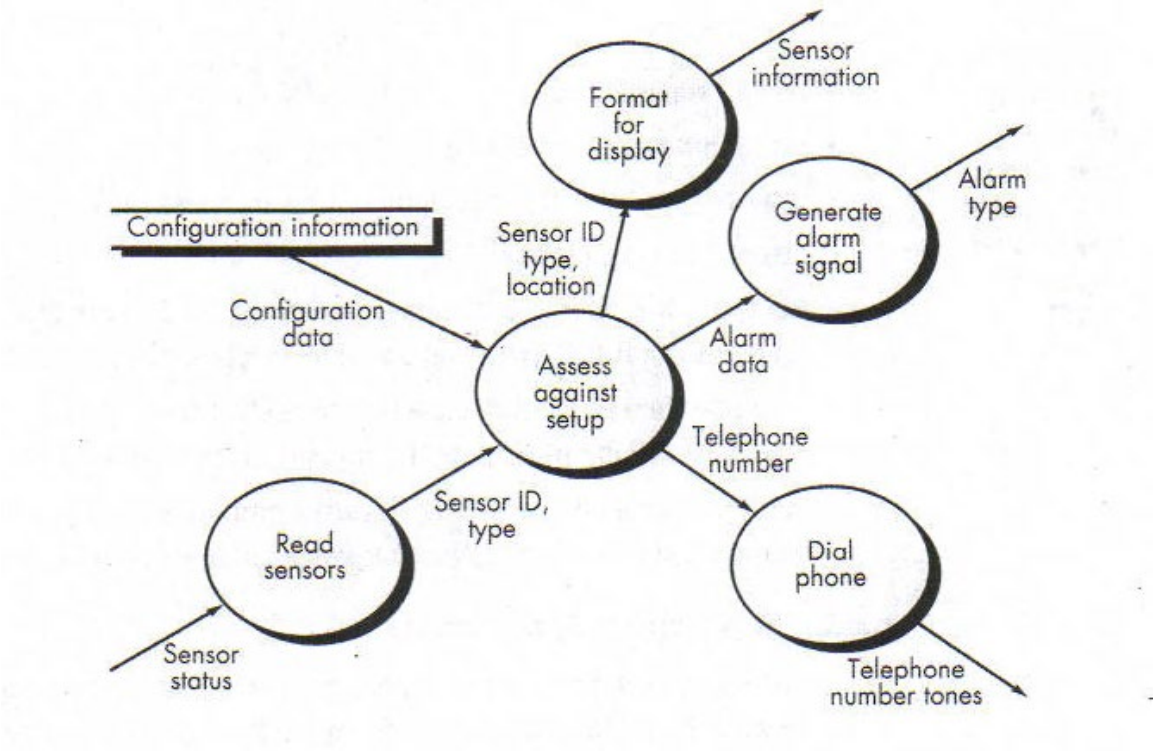
**Level 1 DFD for the safe home security function**




The homeowner receives security information via a control panel, the PC, or a browser, collectively called an interface. The interface displays prompting messages and system status information on the control panel, the PC, or the browser window

The process represented at DFD level 1 can be further refined into lower levels. For example, the process monitor sensors can be refined into a level 2 DFD.

**Level 2 DFD that refines the monitor sensors process**




The refinement of DFDs continues until each bubble performs a single function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component

25. a) Explain about the concept of Software Risk.

**Software Risks**

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics: uncertainty—the risk may or may not happen; that is, there are no 100 percent probable risks1—and loss—if the risk becomes a reality, unwanted consequences or losses will occur. When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development 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 are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance

(or)

b) Illustrate about the Metric of Process.

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.

Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process

improvement. For this reason, many of the same metrics are used in both the process and project domains.

**Process Metrics and Software Process Improvement**

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement (Chapter 37). But before we discuss software metrics and their impact on software process improvement, it is important to

note that process is only one of a number of "controllable factors in improving software quality and organizational performance".

Process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people have been shown to be the most influential factors in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact. 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 before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time used, 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 described in

The skill and motivation of the software people doing the work are the most important factors that influence software quality.

"Software metrics let you know when to laugh and when to cry."

**Process Product People**

**Technology** Development

environment

Customer

characteristics

Business conditions

26. a) Discuss about Risk Projection and Risk Refinement.

**Risk Projection**

Risk projection, also called risk estimation, attempts to rate each risks in two ways—(1) the likelihood or probability that the risk is real and will occur 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.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, you can allocate resources where they will have the most impact.

**Developing a Risk Table**

A risk table provides you with a simple technique for risk projection.

- Begin by listing all risks (no matter how remote) in the first column of the table.
- 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.
- Each risk component is assessed and an impact category is determined.

- The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.
- Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact.
- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom.
- This accomplishes first-order risk prioritization. Assessing Risk Impact

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

- The nature of the risk indicates the problems that are likely if it occurs.
- The scope of a risk combines the severity with its overall distribution

Finally, the timing of a risk considers when and for how long the impact will be felt.

**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.
This general condition can be refined in the following manner:

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

The consequences associated with these refined sub conditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

(or)

b) Explain about RMMM plan.

**The RMMM Plan**

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate risk mitigation, monitoring and management plan.

The RMMM plan documents all work performed as part of risk analysis and are used by the project manager as part of the overall project plan. Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet (RIS). In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to

collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin

**Reg.No** _____

**[17CTU402]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University)**
**(Established Under Section 3 of UGC Act 1956)**
**Coimbatore – 641 021**

**COMPUTER TECHNOLOGY**
**Fourth Semester**
**THIRD INTERNAL EXAMINATION – MARCH 2019**

**SOFTWARE ENGINEERING**

**Class: II B.Sc. CT** **Duration :** 2 Hours
**Date & Session : 11.03.2019 & AN** **Maximum :** 50 Marks

---

**PART-A (20 X 1 = 20 Marks)**
**Answer All the Questions**

1. We should design for direct interaction with _____ that appear on the screen.
   a) code b) class c) objects d) user
2. We should hide technical _____ from the casual user.
   a) reactions b) actions c) internals d) interactions
3. We should streamline _____ as skill levels advance and allow the interaction to be customized.
   a) internals b)interaction c) actions d) reactions
4. Testing should begin in the _____ and progress toward testing in the large.
   a) design b) beginning c)small d) big
5. The less there is to test, the more _____ we can test it.
   a) quickly b) shortly c) automatic d) hardly
6. A good _____ is one that has a high probability of finding an undiscovered error.
   a) planning b)test case c) objective d) goal
7. The design should be _____ for quality as it is being created not after the fact.
   a) reviewed b)assessed c) structured d) integrated
8. The design should be _____ to minimize conceptual errors.
   a) reviewed b) assessed c) structured d) integrated
9. Software design is both a _____ and a model.
   a) model b)process c) data d) function
10. The _____ is the only way that we can accurately translate a customer's requirements into a finished software product or system.
    a) specification b)design c) data d) prototype
11. We should provide _____ interaction.
    a) rigid b)flexible c) encouraging d) enthusiastic
12. The _____ is a process of executing a program with the intent of finding an error.
    a) coding b)testing c) debugging d) designing
13. The interface should present and acquire _____ in a consistent fashion.
    a) information b) task c) knowledge d) idea
14. The interface should present and acquire information in a _____ fashion.
    a ) consistent b) inconsistent c) rigid d) flexible

15. A _____ of the entire system incorporates data, architectural interface, and procedural representations of the software.
    a) data model  b)design model  c) user model  d)system image
16. By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting. This characteristic is called _____.
    a) decomposability  b) simplicity  c) stability  d) understandability
17. The less there is to test, the more quickly we can test it. This characteristic is called _____.
    a) controllability  b)simplicity  c) operability  d) observability
18. The fewer the changes, the fewer the disruptions to testing. This characteristic is called _____.
    a) controllability  b) decomposability  c)stability  d)understandability
19. The _____ focus on the design of the business or technical process that the system must accommodate.
    a) framework models  b) dynamic models  c)process models  d) functional models
20. The _____ task involves the programmer to receive a specification of a module.
    a) Integration Programming  b) System Programming
    c) Unit Testing  d) Configuration Control

**PART-B (3 X 2 = 6 Marks)**
**(Answer All the Questions)**

21. Define architectural design.
22. What is Inheritance?
23. What do you mean by an error?

**PART-C (3 X 8 = 24 Marks)**
**(Answer All the Questions)**

24. a) Write in detail about the approach used in class based components.
    (or)
    b) Discuss in detail about the Architectural components of software.

25. a) Explain Graph based testing methods in Black Box testing.
    (or)
    b) Write in detail about Software Testing Fundamentals.

26. a) Discuss in detail about Validation testing.
    (or)
    b) Illustrate about loop testing and its types.

**Reg.No _____**

**[17CTU402]**

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University)**
**(Established Under Section 3 of UGC Act 1956)**
**Coimbatore – 641 021**

**COMPUTER TECHNOLOGY**
**Fourth Semester**
**THIRD INTERNAL EXAMINATION – MARCH 2019**

**SOFTWARE ENGINEERING**

**Class: II B.Sc. CT** **Duration :** 2 Hours
**Date & Session : 11.03.2019 & AN** **Maximum :** 50 Marks

---

**PART-A (20 X 1 = 20 Marks)**
**Answer All the Questions**

1. We should design for direct interaction with _____ that appear on the screen.
   a) code b) class c) **objects** d) user
2. We should hide technical _____ from the casual user.
   a) reactions b) actions c) **internals** d) interactions
3. We should streamline _____ as skill levels advance and allow the interaction to be customized.
   a) internals b) **interaction** c) actions d) reactions
4. Testing should begin in the _____ and progress toward testing in the large.
   a) design b) beginning c)**small** d) big
5. The less there is to test, the more _____ we can test it.
   a) **quickly** b) shortly c) automatic d) hardly
6. A good _____ is one that has a high probability of finding an undiscovered error.
   a) planning b)**test case** c) objective d) goal
7. The design should be _____ for quality as it is being created not after the fact.
   a) reviewed b)**assessed** c) structured d) integrated
8. The design should be _____ to minimize conceptual errors.
   a) **reviewed** b) assessed c) structured d) integrated
9. Software design is both a _____ and a model.
   a) model b)**process** c) data d) function
10. The _____ is the only way that we can accurately translate a customer's requirements into a finished software product or system.
    a) specification b)**design** c) data d) prototype
11. We should provide _____ interaction.
    a) rigid b)**flexible** c) encouraging d) enthusiastic
12. The _____ is a process of executing a program with the intent of finding an error.
    a) coding b)**testing** c) debugging d) designing
13. The interface should present and acquire _____ in a consistent fashion.
    a) **information** b) task c) knowledge d) idea
14. The interface should present and acquire information in a _____ fashion.
    a ) **consistent** b) inconsistent c) rigid d) flexible

15. A _____ of the entire system incorporates data, architectural interface, and procedural representations of the software.
    a) data model b)**design model** c) user model d)system image
16. By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting. This characteristic is called _____.
    a) **decomposability** b) simplicity c) stability d) understandability
17. The less there is to test, the more quickly we can test it. This characteristic is called _____.
    a) controllability b)**simplicity** c) operability d) observability
18. The fewer the changes, the fewer the disruptions to testing. This characteristic is called _____.
    a) controllability b) decomposability c)**stability** d)understandability
19. The _____ focus on the design of the business or technical process that the system must accommodate.
    a) framework models b) dynamic models c)**process models** d) functional models
20. The _____ task involves the programmer to receive a specification of a module.
    a) Integration Programming b) System Programming
    c) **Unit Testing** d) Configuration Control

**PART-B (3 X 2 = 6 Marks)**
**(Answer All the Questions)**

21. Define architectural design.

    The *architectural design* defines the relationship between more structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which the architectural design can be implemented.

    The *architectural design* can be derived from the System Specs, the analysis model, and interaction of subsystems defined within the analysis model.

22. What is Inheritance?

    **Inheritance**

Design options:

- The class can be designed and built from scratch. That is, inheritance is not used.
- The class hierarchy can be searched to determine if a class higher in the hierarchy (a super-class) contains most of the required attributes and operations.
- The new class inherits from the super-class and additions may then be added, as required.

- The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
- Characteristics of an existing class can be overridden and different versions of attributes or operations are implemented for the new class.

23. What do you mean by an error?

Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.

1. Testing is a process of executing a program with the intent of finding an error.

2. A good test case is one that has a high probability of finding an as-yet undiscovered error.

3. A successful test is one that uncovers an as-yet-undiscovered error.

**PART-C (3 X 8 = 24 Marks)**
**(Answer All the Questions)**

24. a) Write in detail about the approach used in class based components.

Class-based Component Design

- Focuses on the elaboration of domain specific analysis classes and the definition of infrastructure classes
- Detailed description of class attributes, operations, and interfaces is required prior to beginning construction activities

Class-based Component Design Principles

- Open-Closed Principle (OCP) – class should be open for extension but closed for modification
- Liskov Substitution Principle (LSP) – subclasses should be substitutable for their base classes • Dependency Inversion Principle (DIP) – depend on abstractions, do not depend on concretions
- Interface Segregation Principle (ISP) – many client specific interfaces are better than one general purpose interface
- Release Reuse Equivalency Principle (REP) – the granule of reuse is the granule of release • Common Closure Principle (CCP) – classes that change together belong together
- Common Reuse Principle (CRP) – Classes that can't be used together should not be grouped together

Component-Level Design Guidelines

- Components
  - Establish naming conventions in during architectural modeling
  - Architectural component names should have meaning to stakeholders
  - Infrastructure component names should reflect implementation specific meanings ○ Use of stereotypes may help identify the nature of components

- Interfaces
    - Use lollipop representation rather than formal UML box and arrow notation
    - For consistency interfaces should flow from the left-hand side of the component box ○ Show only the interfaces relevant to the component under construction
- Dependencies and Inheritance
    - For improved readability model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)
    - Component interdependencies should be represented by interfaces rather that component to component dependencies

**Cohesion (lowest to highest)**

- Utility cohesion – components grouped within the same category but are otherwise unrelated • Temporal cohesion – operations are performed to reflect a specific behavior or state
- Procedural cohesion – components grouped to allow one be invoked immediately after the preceding one was invoked with or without passing data
- Communicational cohesion –operations required same data are grouped in same class
- Sequential cohesion – components grouped to allow input to be passed from first to second and so on
- Layer cohesion – exhibited by package components when a higher level layer accesses the services of a lower layer, but lower level layers do not access higher level layer services

- Functional cohesion – module performs one and only one function

**Coupling**

- Content coupling – occurs when one component surreptitiously modifies internal data in another component
- Common coupling – occurs when several components make use of a global variable
- Control coupling – occurs when one component passes control flags as arguments to another • Stamp coupling – occurs when parts of larger data structures are passed between components • Data coupling – occurs when long strings of arguments are passed between components
- Routine call coupling – occurs when one operator invokes another
- Type use coupling – occurs when one component uses a data type defined in another

- Inclusion or import coupling – occurs when one component imports a package or uses the content of another

- External coupling – occurs when a components communications or collaborates with infrastructure components (e.g. database)

Conducting Component-Level Design

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.

3. Elaborate all design classes that are not acquired as reusable components. a. Specify message details when classes or components collaborate. b. Identify appropriate interfaces for each component.
   c. Elaborate attributes and define data types and data structures required to implement them.
   d. Describe processing flow within each operation in detail.
4. Identify persistent data sources (databases and files) and identify the classes required to manage them.
5. Develop and elaborate behavioral representations for each class or component.
6. Elaborate deployment diagrams to provide additional implementation detail.
7. Refactor every component-level diagram representation and consider alternatives.

(or)

b) Discuss in detail about the Architectural components of software.

Architectural design

- The architectural design starts then the developed software is put into the context.
- The information is obtained from the requirement model and other information collect during the requirement engineering.

**Representing the system in context**

All the following entities communicates with the target system through the interface that is small rectangles shown in above figure.

**Superordinate system**
These system use the target system like a part of some higher-level processing scheme.

**Subordinate system**
This systems is used by the target system and provide the data mandatory to complete target system functionality.

**Peer-level system**
These system interact on peer-to-peer basis means the information is consumed by the target system and the peers.

**Actors**
These are the entities like people, device which interact with the target system by consuming information that is mandatory for requisite processing.

Defining Archetype

- An archetype is a class or pattern which represents a core abstraction i.e critical to implement or design for the target system.
- A small set of archetype is needed to design even the systems are relatively complex.

- The target system consists of archetype that represent the stable elements of the architecture. • Archetype is instantiated in many different forms based on the behavior of the system.
- In many cases, the archetype is obtained by examining the analysis of classes defined as a part of the requirement model.

An Architecture Trade-off Analysis Method (ATAM)

ATAM was developed by the Software Engineering Institute (SEI) which started an iterative evaluation process for software architecture.

**The design analysis activities which are executed iteratively that are as follows:**

**1. Collect framework**
Collect framework developed a set of use cases that represent the system according to user point of view.

**2. Obtained requirement, Constraints, description of the environment.**
These types of information are found as a part of requirement engineering and is used to verify all the stakeholders are addressed properly.

**3. Describe the architectural pattern**
The architectural patterns are described using an architectural views which are as follows:

**Module view:** This view is for the analysis of assignment work with the components and the degree in which abstraction or information hiding is achieved

**Process view:** This view is for the analysis of the software or system performance.

**Data flow view:** This view analyzes the level and check whether functional requirements are met to the architecture.

**4. Consider the quality attribute in segregation**
The quality attributes for architectural design consist of reliability, performance, security, maintainability, flexibility, testability, portability, re-usability etc.

**5. Identify the quality attributes sensitivity**

- The sensitivity of quality attributes achieved by making the small changes in the architecture and find the sensitivity of the quality attribute which affects the performance.
- The attributes affected by the variation in the architecture are known as sensitivity points.

25. a) Explain Graph based testing methods in Black Box testing.

**Graph-Based Testing Methods**

The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects.

Next step is to define a series of tests that verify "all objects have the expected relationship to one another".

To accomplish these steps, the software engineer begins 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.

**Fig (A) Graph notation (B) Simple example**

The symbolic representation of a graph is shown in Fig A.

Nodes are represented as circles connected by links that take a number of different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction.

A bidirectional link, called a **symmetric link**, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.

Eg. consider a portion of a graph for a word-processing application (Fig B) where

Object #1 = new file menu select

Object #2 = document window

Object #3 = document text

Referring to the figure, a menu select on new file generates a document window.

The node weight of document window provides a list of the window attributes that are to be expected when the window is generated.

The link weight indicates that the window must be generated in less than 1.0 second.

An undirected link establishes a symmetric relationship between the new file menu select and document text, and parallel links indicate relationships between document window and document text.

In reality, a far more detailed graph would have to be generated as a precursor to test case design.

The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

Beizer describes a number of behavioral testing methods that can make use of graphs:

**Transaction flow modeling**. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., flight. information. input is followed by validation/availability. processing).

**Finite state modeling**. The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., order-information is verified during inventory-availability look-up and is followed by customer-billing-information input). The state transition diagram can be used to assist in creating graphs of this type.

**Data flow modelin**g. The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node FICA.tax.withheld (FTW) is computed from gross.wages (GW) using the relationship, FTW = 0.62 - GW.

**Timing modeling**. The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.

Once nodes have been identified, links and link weights should be established.

In general, links should be named, although links that represent control flow between program objects need not be named.

Each relationship is studied separately so that test cases can be derived.

The transitivity of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, X, Y, and Z. Consider the following relationships:

X is required to compute Y

Y is required to compute Z

Therefore, a transitive relationship has been established between X and Z:

X is required to compute Z

Based on this transitive relationship, tests to find errors in the calculation of Z must consider a variety of values for both X and Y.

The symmetry of a relationship (graph link) is also an important guide to the design of test cases.

As test case design begins, the first objective is to achieve node coverage. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, link coverage is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present.

A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked

(or)

b) Write in detail about Software Testing Fundamentals.

**SOFTWARE TESTING FUNDAMENTALS:**

- Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.
- Software engineers are by their nature constructive people.
- Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.
- Beizer describes this situation effectively when he states: There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt.

**Testing Objectives**

Glen Myers states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software.

Also testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.

In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.

But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

**Testing Principles**

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis [DAV95] suggests a set of testing principles.

**All tests should be traceable to customer requirements**.

The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

**Tests should be planned long before testing begins**.

Test planning can begin as soon as the requirements model is complete.

Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

**The Pareto principle applies to software testing**.

Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

**Testing should begin "in the small" and progress toward testing "in the large."**

The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

**Exhaustive testing is not possible**

The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

**To be most effective, testing should be conducted by an independent third party.**

**Testability**

- Software testability is simply how easily a computer program can be tested.
- Since testing is so profoundly difficult, it pays to know what can be done to streamline it.
- Sometimes programmers are willing to do things that will help the testing process and a checklist of possible design points, features, etc., can be useful in negotiating with them.
- "Testability" occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult. The **checklist** that follows provides a set of characteristics that lead to testable software.

**Operability**. "The better it works, the more efficiently it can be tested."

• The system has few bugs (bugs add analysis and reporting overhead to the test process).

• No bugs block the execution of tests.

• The product evolves in functional stages (allows simultaneous development and testing).

**Observability**. "What you see is what you test."

• Distinct output is generated for each input.

    - System states and variables are visible or queriable during execution.
    - Past system states and variables are visible or queriable (e.g., transaction logs).
    - All factors affecting the output are visible.
    - Incorrect output is easily identified.
    - Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

**Controllability**. "The better we can control the software, the more the testing can be automated and optimized."

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

**Decomposability.** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

- The software system is built from independent modules.
- Software modules can be tested independently.

**Simplicity**. "The less there is to test, the more quickly we can test it."

- Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements).
- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

**Stability.** "The fewer the changes, the fewer the disruptions to testing."

• Changes to the software are infrequent.

• Changes to the software are controlled.

• Changes to the software do not invalidate existing tests.

• The software recovers well from failures.

**Understandability**. "The more information we have, the smarter we will test."

• The design is well understood.

• Dependencies between internal, external, and shared components are well understood.

• Changes to the design are communicated.

• Technical documentation is instantly accessible.

• Technical documentation is well organized.

• Technical documentation is specific and detailed.

• Technical documentation is accurate.

26. a) Discuss in detail about Validation testing.

## VALIDATION TESTING

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between different software categories disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a Software Requirements Specification has been developed, it describes all user-visible attributes of the software and contains a Validation Criteria section that forms the basis for a validation-testing approach.

### i) Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. 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). If a deviation from specification is uncovered, a deficiency list is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

**ii) Configuration Review**

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

**ii) Alpha and Beta Testing**

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be used; output that seemed clear to the tester may be unintelligible to a user in the field. When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time. If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

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

**The beta test** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base. 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 from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

(or)

b) Illustrate about loop testing and its types.

**Loop Testing**

Loops are the cornerstone for the vast majority of all algorithms implemented in software.

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 (Fig).

**imple 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.

4. m passes through the loop where m < n.

5. n -1, n, n + 1 passes through the loop.

**Nested loops**.

If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.

Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.

**Concatenated loops.**

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

**Unstructured loops**.

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.