

**IMPORTANT QUESTIONS AND ANSWERS**

**Department of Computer Science & Engineering**

**SUBJECT CODE: CS8494**

**SUBJECT NAME: SOFTWARE ENGINEERING**

**Regulation: 2017**

**Semester and Year: 04 and II**

## REGULATION 2017

## CS8494

# SOFTWARE ENGINEERING

**LTPC**

**3 0 0 3**

## UNIT I

## SOFTWARE PROCESS AND PROJECT MANAGEMENT

9

## UNIT II

## REQUIREMENTS ANALYSIS AND SPECIFICATION

9

## UNIT III

## SOFTWARE DESIGN

9

## UNIT IV

## TESTING AND IMPLEMENTATION

9

## UNIT V

## PROJECT MANAGEMENT

9

downloaded from [www.rejinpaul.com](http://www.rejinpaul.com)

**TEXT BOOK:**

1. Roger S. Pressman, "Software Engineering – A Practitioner's Approach", Seventh Edition, Mc Graw-Hill International Edition, 2010.

**REFERENCES:**

1. Ian Sommerville, "Software Engineering", 9th Edition, Pearson Education Asia, 2011.

2. Rajib Mall, "Fundamentals of Software Engineering", Third Edition, PHI Learning Private Limited, 2009.

3. Pankaj Jalote, "Software Engineering, A Precise Approach", Wiley India, 2010.

**TABLE OF CONTENTS**

<b>Sl. No.</b>	<b>Topic</b>	<b>Page No.</b>
a.	Aim and Objective of the Subject	1
b.	Detailed Lesson Plan	2
c.	<b>Unit-1</b> Part – A	4
d.	<b>Unit-1</b> Part- B	6
1.	Software life cycle models	6
2.	COCOMO model	15
3.	Estimation	18
4.	Project scheduling	24
5.	Risk management	29
e.	<b>Unit-1</b> Part- C	33
f.	<b>Unit-2</b> Part - A	37
g.	<b>Unit-2</b> Part - B	40
6.	Functional & non-functional requirements	40
7.	Software requirement system	45
8.	Requirements engineering activities	51
9.	Requirement engineering process	52
10.	Components of software requirements	58
11.	Classical analysis with examples.	60
h.	<b>Unit-2</b> Part - C	67
i.	<b>Unit-3</b> Part - A	75
j.	<b>Unit-3</b> Part - B	78
12.	Design Process & Design Concepts	78
13.	Architectural Styles and Design	81
14.	Transform and Transactional Mapping	87
15.	Cohesion and Coupling	94
16.	User Interface Design	96
17.	Design Heuristics	101

k.	<b>Unit-3 Part - C</b>	102
l.	<b>Unit-4 Part - A</b>	108
m.	<b>Unit-4 Part - B</b>	110
18.	White Box testing	110
19.	Black box Testing Methods	114
20.	Various Testing Strategy	119
21.	Software Testing Principles	125
22.	Sample problem for cyclomatic complexity	128
23.	Validation Testing	131
24.	Debugging Process & Coding Process	132
25.	Refactoring	134
n.	<b>Unit-4 Part - C</b>	136
o.	<b>Unit-5 Part - A</b>	144
p.	<b>Unit-5 Part - B</b>	146
26.	Process & project metrics	146
27.	Categories of Software Risks	150
28.	Function Point Analysis	156
29.	COCOMO II Model	158
30.	Software Project Planning	161
31.	i)Project Scheduling ii) Timeline Charts	164
32.	Problem-Based Estimation	169
q.	<b>UNIT-5 PART - C</b>	172
r.	Industry Connectivity and Latest Developments	178
s.	<b>University Question papers</b>	179

## AIM AND OBJECTIVE OF THE SUBJECT

The student should be made to:

1. Understand the phases in a software project
2. Understand fundamental concepts of requirements engineering and Analysis Modeling.
3. Understand the major considerations for enterprise integration and deployment.
4. Learn various testing and maintenance measures

## DETAILED LESSON PLAN

### Text Book

1. Roger S. Pressman, "Software Engineering – A Practitioner's Approach", Seventh Edition, Mc Graw-Hill International Edition, 2010.

### References

1. Ian Sommerville, "Software Engineering", 9th Edition, Pearson Education Asia,
2. Rajib Mall, "Fundamentals of Software Engineering", Third Edition, PHI Learning Private Limited, 2009.
3. Pankaj Jalote, "Software Engineering, A Precise Approach", Wiley India, 2010.
4. Kelkar S.A., "Software Engineering", Prentice Hall of India Pvt Ltd, 2007.
5. Stephen R.Schach, "Software Engineering", Tata McGraw-Hill Publishing Company Limited, 2007.

Sl. No	Unit	Topic / Portions to be Covered	Hours Required / Planned	Cumulative Hrs	Books Referred
<b>UNIT – 1- SOFTWARE PROCESS AND PROJECT MANAGEMENT</b>					
1	1	Introduction to Software Engineering	1	1	T1
2	1	Software Process	1	2	T1
3	1	Perspective and Specialized Process Models	2	4	T1
4	1	Software Project Management: Estimation	1	5	T1
5	1	LOC and FP Based Estimation	1	6	T1
6	1	COCOMO Model	1	7	R1
7	1	Project Scheduling, Scheduling	1	8	T1
8	1	Earned Value Analysis	1	9	T1
9	1	Risk Management	1	10	T1
<b>UNIT – II- REQUIREMENTS ANALYSIS AND SPECIFICATION</b>					
10	2	Software Requirements: Functional and Non-Functional	1	11	R1
11	2	User requirements, System requirements	1	12	R1
12	2	Software Requirements Document	1	13	R1
13	2	Requirement Engineering Process: Feasibility Studies	1	14	R1
14	2	Requirements elicitation and analysis	1	15	R1
15	2	requirements validation	1	16	R1
16	2	requirements management	1	17	R1
17	2	Classical analysis	1	18	R1
18		Structured system Analysis	1	19	R1
19	2	Petri Nets- Data Dictionary	1	20	WEB
<b>UNIT – III- SOFTWARE DESIGN</b>					
20	3	Design process, Design Concepts	1	21	T1
21	3	Design Model, Design Heuristic	1	22	T1

Sl. No	Unit	Topic / Portions to be Covered	Hours Required / Planned	Cumulative Hrs	Books Referred
22	3	Architectural Design	1	23	T1
23	3	Architectural styles	1	24	T1
24	3	Architectural Mapping using Data Flow	2	26	T1
25	3	User Interface Design: Interface analysis	1	27	T1
26	3	Interface Design	1	28	T1
27	3	Component level Design: Designing Class based components	1	29	T1
28	3	Traditional Components	1	30	T1

#### UNIT – IV- TESTING AND IMPLEMENTATION

29	4	Software testing fundamentals, Internal and external views of Testing	1	31	T1
30	4	White box testing	1	32	T1
31	4	Basis path testing-control structure testing	1	33	T1
32	4	Black box testing	2	35	T1
33	4	Regression Testing	1	36	T1
34	4	Unit Testing	1	37	T1
35	4	Integration Testing	1	38	T1
36	4	Validation Testing	1	39	T1
37	4	System Testing And Debugging	1	40	T1
38	4	Software Implementation Techniques: Coding practices, Refactoring	1	41	T1

#### UNIT – V- PROJECT MANAGEMENT

39	5	Estimation: FP Based, LOC Based, Make/Buy Decision	1	42	T1
40	5	COCOMO II	1	43	T1
41	5	Planning , Project Plan, Planning Process	1	44	R1
42	5	RFP Risk Management, Identification , Projection, RMMM	1	45	T1
43	5	Scheduling and Tracking	1	46	T1
44	5	Relationship between people and effort, Task Set & Network	1	47	T1
45	5	EVA	1	48	T1
46	5	Process and Project Metrics	2	50	T1

## UNIT – 1 SOFTWARE PROCESS AND PROJECT MANAGEMENT

Introduction to Software Engineering, Software Process, Perspective and Specialized Process Models – Software Project Management: Estimation – LOC and FP Based Estimation, COCOMO Model – Project Scheduling – Scheduling, Earned Value Analysis - Risk Management.

### PART-A

#### 1. What are the two types of software Products?

**May: 12**

1. Generic: These products are developed and to be sold to the range of different customers.
2. Custom: These types of products are developed and sold to the specific group of customers and developed as per their requirements

#### 2. What is software Engineering?

**Dec: 13**

Software Engineering is a discipline in which theories, methods and tools are applied to develop professional software product.

#### 3. What is software process?

**May: 13**

Software process can be defined as the structured set of activities that are required to develop the software system. The fundamental activities are

1. Specification
2. Design and implementation
3. Validation
4. Evolution.

#### 4. Write the process framework & umbrella activities

**May: 15**

Process Framework Activities: Communication, Planning, Modeling, Construction, Deployment.

Umbrella Activities:

1. Software Project Tracking and Control
2. Risk Management
3. Software Quality Assurance
4. Formal Technical reviews
5. Software Configuration Management
6. Work Product Preparation and Production
7. Reusability Management
8. Measurement

#### 5. State the advantages and disadvantages in LOC based cost estimation.

**May : 15**

Advantages: 1. Artifact of software development which is easily counted.

2. Many existing methods use LOC as a key input.

3. A large body of literature and data based on LOC already exists.

Disadvantages: 1. This measure is dependent upon the programming language.

2. This method is well designed but shorter program may get suffered.

3. It does not accommodate non procedural languages.

**6. Software doesn't wear out. Justify****Dec: 13**

1. Software does not get affected from environmental maladies such as temperature, dusts and vibrations. But due to some undiscovered errors the failure rate is high and drops down as soon as the errors get corrected.
2. Another issue with software is that there are no spare parts for software. If hardware component wears out it can be replaced by another component but it is not possible in case of software.

**7. Define estimation****May 05, 06, 07, Dec 07, 10**

Software project estimation is a form of problem solving. Many times the problem to be solved is too complex in software engineering. Hence for solving such problems, we decompose the given problem into a smaller problem.

The decomposition can be done using two approaches: decomposition of problem or decomposition of process. Estimation uses one or both forms of decomposition.

**8. Differentiate between size oriented and function oriented metrics. May: 13**

Sl.No	Size Oriented Metrics	Function Oriented Metrics
1.	Size oriented software metrics is by considering the size of the software that has been produced.	Function oriented metrics use a measure of functionality delivered by the software.
2.	For a size oriented metric the software organization maintains simple records in tabular form. The typical table entries are : Project Name , LOC , Effort , Errors, Defects	Most widely used function oriented metric as the Function Point. Computation of the function point is based on characteristics of software, information domain and complexity.

**9. An organic software occupies 15,000 LOC. How many programmers are needed to complete? Dec : 12**

Ans: System = Organic

LOC = 15KLOC

$E = a_b(KLOC)b_b$

$= 2.4(15)^{1.05}$

$= 41 \text{ persons} - \text{month}$

**10. What is risk? Give an example of risk.****Dec: 13**

The software risk can be defined as a problem that can cause some loss or can threaten the success of the project.

Eg: Experienced and skilled staff leaving the organization in between.

**11. Define system engineering.**

Systems engineering is an interdisciplinary approach to design, implementation and evaluation that holds the key to the successful development of complex human-made systems.

**12. Depict the relationship between work product task activity and system**  
**Nov :16**

**Work product :** A work product may begin as an analysis made during the development of a project, creating a type of proposal for the project that a company cannot deliver until the project has received approval.

**Task :** A task can also be an activity that is being imposed on another person or a demand on the powers of another. A task can be said to be any activity that is done with a particular purpose.

**Activity** An activity is a major unit of work to be completed in achieving the objectives of a process. An activity has precise starting and ending dates, incorporates a set of tasks to be completed, consumes resources, and results in work products. An activity may have a precedence relationship with other activities.

**System :** A set of detailed methods, procedures and routines created to carry out a specific activity, perform a duty, or solve a problem.

**13. What led to the transition from product oriented development to process oriented development?**  
**May : 16**

The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. It is a standalone entity that can be produced by development organization.

The process can be defined as the structured set of activities that are required to develop software system.

**14. Mention the characteristics of software contrasting it with characteristics of hardware.**  
**May : 16**

- Software is nothing but a collection of computer programs that are related documents that are indented to provide desired features, functionalities and better performance.
- Software is engineered not manufactured.
- Software does not wear out.
- Most software is custom built rather than being assembled from components.

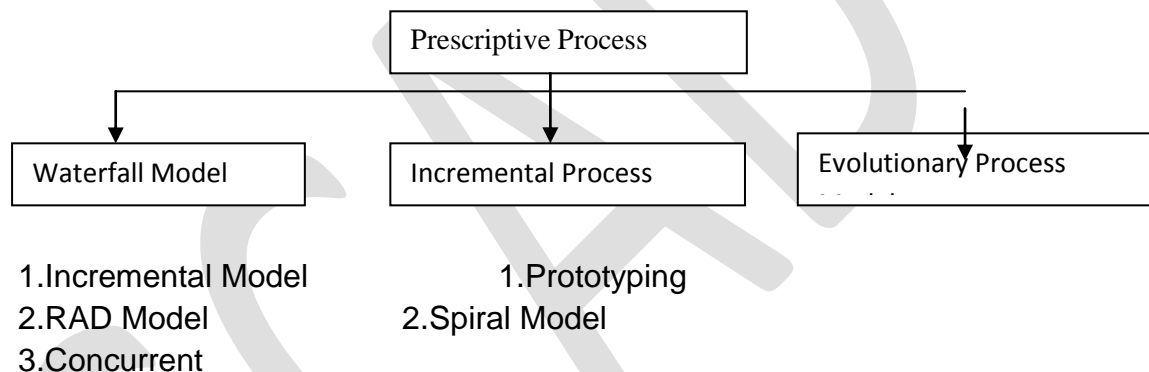
**15. If you have to develop a word processing software product ,what process model will use you choose? Justify your answer.**  
**Nov :16**

Prototype model will be used in development of this application. This model is made in series of increment throughout project. Prototype model is strategy that allows system to be developed in pieces. It allows the additions in process as per requirements, process change can be implemented.

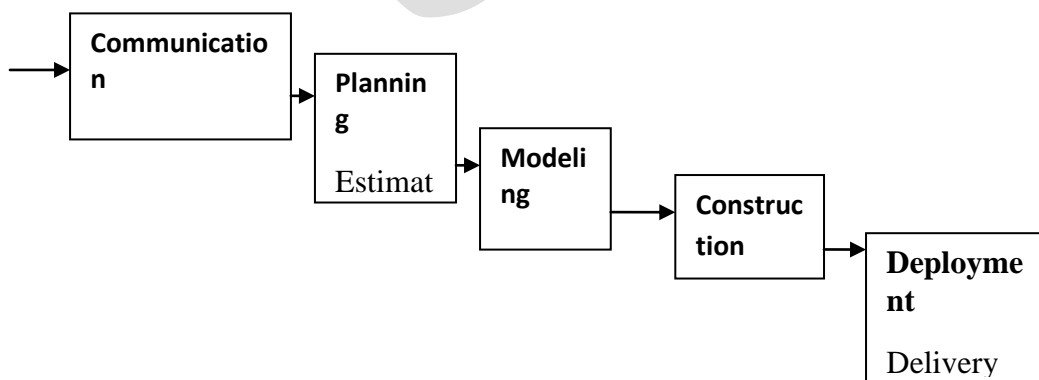
**PART-B****SOFTWARE LIFE CYCLE MODELS**

**1. Neatly explain all the Prescriptive process models and Specialized process models** May: 03, 05, 06,09,10,14,16 Dec : 04,08,09,12 ,16

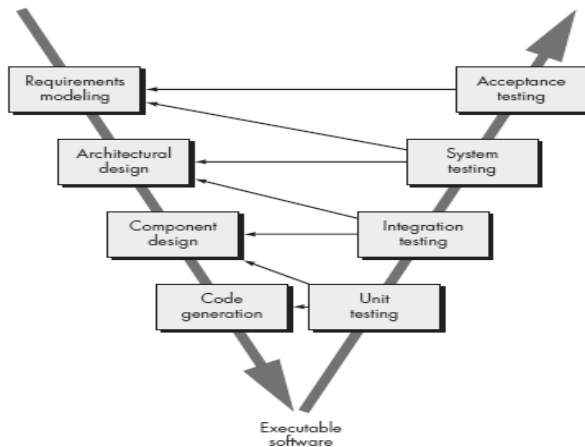
- “Prescriptive” means a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements are interrelated to one another.
- The software process model is also known as Software Development Life Cycle (SDLC) Model for or software paradigm.
- Various prescriptive process models are

**Need for Process Model**

Each team member in software product development will understand –what is the next activity and how to do it. Thus process model will bring the definiteness and discipline in overall development process.

**1. The Waterfall Model**

The waterfall model, sometimes called the classic life cycle, is a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software. A variation in the representation of the waterfall model is called the V-model. Represented in figure, the V-model depicts the relationship of quality assurance actions to the actions associated with



Communication, modeling, and early construction activities. As software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests that validate each of the models created as the team moved down the left side.

The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work. The waterfall model is the oldest paradigm for software engineering.

Disadvantages:

1. It is difficult to follow the sequential flow in software development process. If some changes are made at some phases then it may cause confusion.
2. The requirement analysis is done initially and sometimes it is not possible to state all the requirements explicitly in the beginning. This causes difficulty in the projects.
3. The customer can see the working model of the project only at the end. After reviewing of the working model; if the customer gets dissatisfied then it causes serious problem.

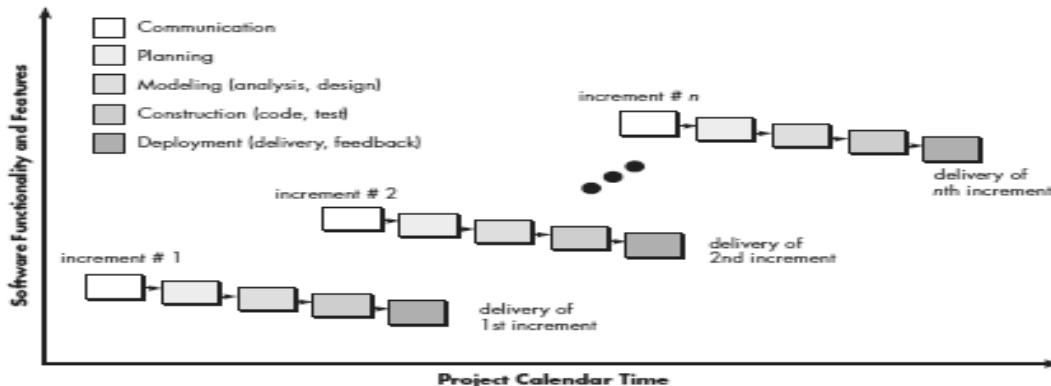
The waterfall model can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

## 2. Incremental Process Models

The incremental model delivers series of releases to the customer. These releases are called increments.

- A process model that is designed to produce the software in increments. The incremental model combines elements of linear and parallel process flows the

incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.



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

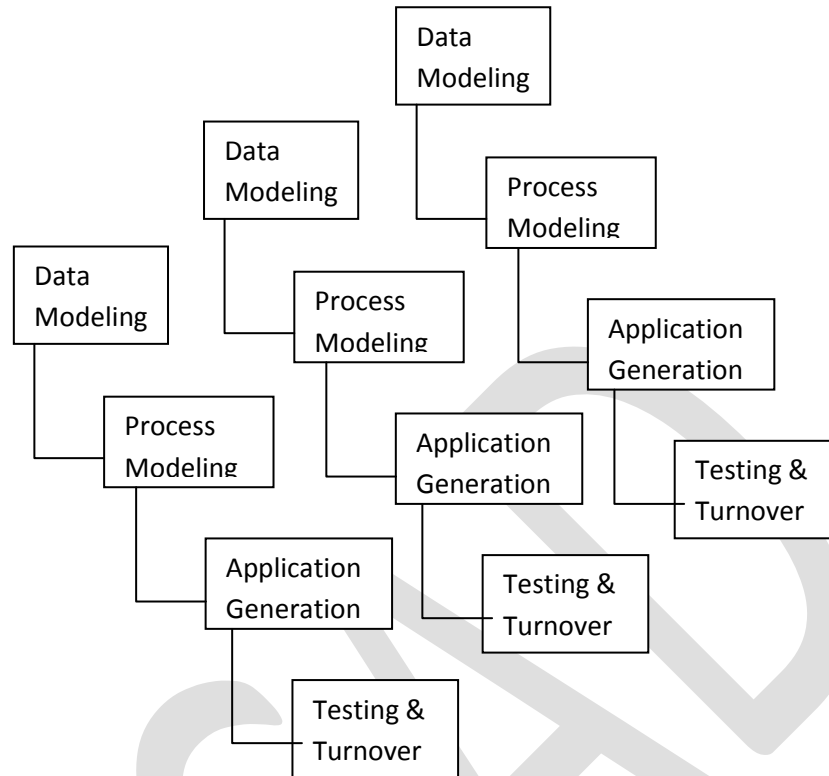
#### Advantages

- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people.

#### i).RAD Model

RAD model is Rapid Application Development model. It is a type of incremental model. In RAD model the components or functions are developed in parallel as if they were mini projects. The developments are time boxed, delivered and then assembled into a working prototype. This can quickly give the customer

something to see and use and to provide feedback regarding the delivery and their requirements.



**Business modeling:** The information flow is identified between various business functions.

**Data modeling:** Information gathered from business modeling is used to define data objects that are needed for the business.

**Process modeling:** Data objects defined in data modeling are converted to achieve the business information flow to achieve some specific business objective. Description are identified and created for CRUD of data objects.

**Application generation:** Automated tools are used to convert process models into code and the actual system.

**Testing and turnover:** Test new components and all the interfaces.

**Advantages of the RAD model:**

- Reduced development time.
- Increases reusability of components
- Quick initial reviews occur
- Encourages customer feedback
- Integration from very beginning solves a lot of integration issues.

### Disadvantages of RAD model:

- Depends on strong team and individual performances for identifying business requirements.
- Only system that can be modularized can be built using RAD
- Requires highly skilled developers/designers.
- High dependency on modeling skills
- Inapplicable to cheaper projects as cost of modeling and automated code generation is very high.

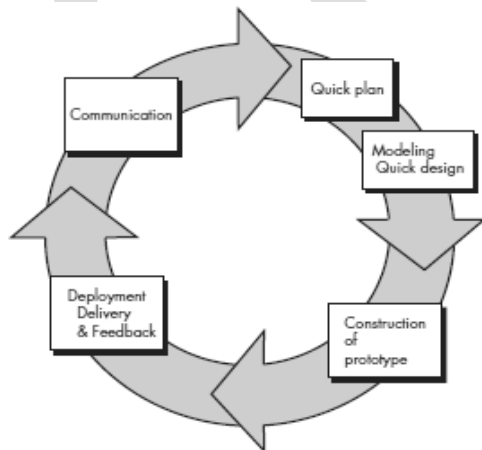
### When to use RAD model:

- RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
- It should be used if there's high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
- RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

### 3. Evolutionary Process Models

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software two common evolutionary process models.

**i) Prototyping:** Prototyping can be used as a stand-alone process model; it is more commonly used as a technique that can be implemented within the context of any one of the process models. The prototyping paradigm assists the stakeholders to better understand what is to be built when requirements are fuzzy.



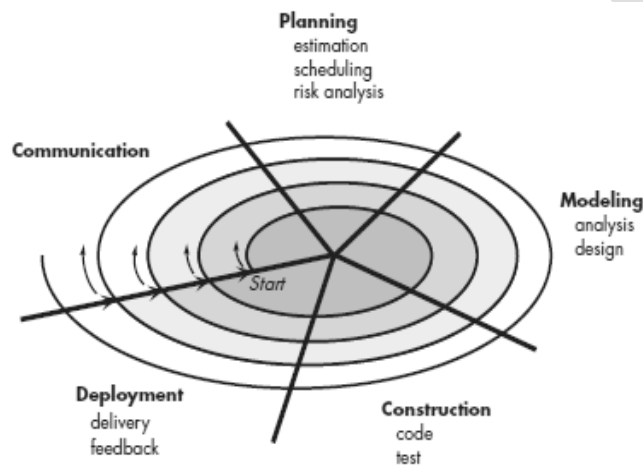
The prototyping paradigm begins with communication. We meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. Prototyping iteration is planned quickly, and modeling occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype.

The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done. The prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.

**Disadvantages:**

1. In the first version itself, customer often wants “few fixes” rather than rebuilding of the system whereas rebuilding of new system maintains high level of quality.
2. Sometimes developer may make implementation compromises to get prototype working quickly. Later on developer may become comfortable with compromises and forget why they are inappropriate.

**ii) The Spiral Model.**



Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.

It has two main distinguishing features.

i) One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.

ii) The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions. Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, each of the framework activities represents one segment of the spiral path. Risk is considered as each revolution is

made. 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; 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. The project manager adjusts the planned number of iterations required to complete the software.

#### Advantages

- The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.

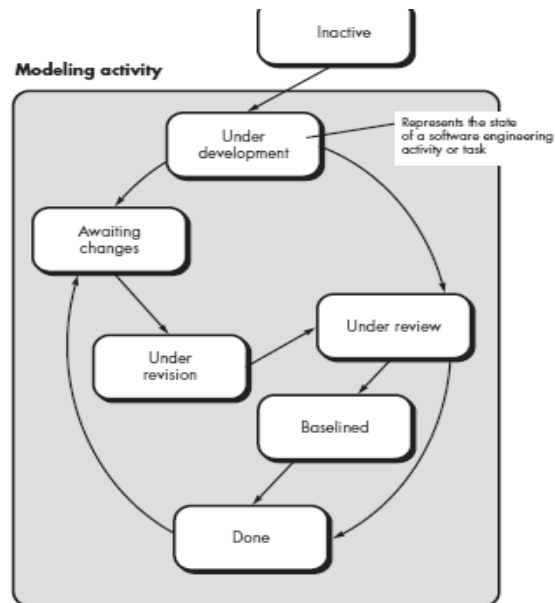
### iii) Concurrent Models

The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models. Figure provides a schematic representation of one software engineering activity.

The activity—modeling—may be in any one of the states<sup>12</sup> noted at any given time. Similarly, other activities, actions, or tasks (e.g., communication or construction) can be represented in an analogous manner. For example, early in a project the communication activity has completed its first iteration and exists in the awaiting changes state.

The modeling activity (which existed in the inactive state) while initial communication was completed, now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the under development state into the awaiting changes state.

Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. This generates the event analysis model correction, which will trigger the requirements analysis action from the done state into the awaiting changes state.



## Specialized process models

### A. Component-Based Development

Component-based development model constructs applications from prepackaged software components. 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 or packages of classes.

Component-based development model incorporates the following steps

1. Component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. 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.

### B. The Formal Methods Model

Formal methods enable you 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. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

- The development of formal models is currently quite time consuming and expensive.

- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

### C. Aspect-Oriented Software Development

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—“mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern” calls aspect-oriented component engineering (AOCE):

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. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on.

## COCOMO MODEL

### 2. What is COCOMO Model? Explain in detail.

May: 07, 08, 14, Dec:05,13

Stands for Constructive Cost Model.

COCOMO is one of the most widely used software estimation models in the world.

- Introduced by Barry Boehm in 1981 in his book “Software Engineering Economics”
- Became one of the well-known and widely-used estimation models in the industry.
- It has evolved into a more comprehensive estimation model called COCOMO II .

#### COCOMO Cost Drivers

- Personnel Factors
- Applications experience
- Programming language experience

Product Factors	Platform Factors	Project Factors
• Required software reliability	• Execution time constraint	• Use of software tools
• Database size	• Main storage constraint	• Use of modern programming practices
• Software product complexity	• Computer turn-around time	• Required development schedule
• Required reusability	• Virtual machine volatility	• Classified security application
• Documentation match to life cycle needs	• Platform volatility	• Multi-site development
• Product reliability and complexity	• Platform difficulty	• Requirements volatility

COCOMO has three different models that reflect the complexity

Model 1. The Basic COCOMO model is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code (LOC).

Model 2. The Intermediate COCOMO model computes software development effort as a function of program size and a set of "cost drivers" that include subjective assessments of product, hardware, personnel and project attributes.

Model 3. The Advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

The COCOMO models are defined for three classes of software projects. Using Boehm's terminology these are: (1) organic mode—relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements (e.g., a thermal analysis program developed for a heat transfer group);

(2) Semi-detached mode —an intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements (e.g., a transaction processing system with fixed requirements for terminal hardware and data base software);

(3) Embedded mode —a software project that must be developed within a set of tight hardware, software and operational constraints (e.g., flight control software for aircraft).

Let us understand each model in detail:

1. Basic Model: The basic COCOMO model estimates the software development effort using only Lines of Code .Various Equations in this model are

$$E = a_b \text{ KLOC } b_b$$

$$D = c_b E^{d_b}$$

$$P = E/D$$

Where E is the effort applied in person – month. D is the development time in chronological months.

KLOC means kilo lines of code for the project. The coefficients  $a_b$  ,  $b_b$  ,  $c_b$  ,  $d_b$  for the

Table 1. Basic COCOMO Model

Software Project	$a_b$	$b_b$	$c_b$	$d_b$
organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
embedded	3.6	1.20	2.5	0.32

2) The intermediate COCOMO model takes the form:

The basic model is extended to consider a set of "cost driver attributes" that can be grouped into four major categories:

1.Product attributes	2.Hardware attributes	3.Personnel attributes	4. Project attributes
a. required software reliability	a. run-time performance constraints	a. analyst capability	a. use of software tools
b. size of application data base	b. memory constraints	b. Software engineer capability	b. Application of software engineering methods
c. complexity of the product	c. volatility of the virtual machine environment	c. Applications experience	c. Required development schedule
	d. required turnaround time	d. virtual machine experience	
		e. programming language experience	

Each of the 15 attributes is rated on a 6 point scale that ranges from "very low" to "extra high". Based on the rating, an effort multiplier is determined from tables published by Boehm [BOE81], and the product of all effort multipliers results is an effort adjustment factor (EAF). Typical values for EAF range from 0.9 to 1.4.

$$E = a_i KLOC^{b_i} \times EAF \text{ person-months}$$

where  $E$  is the effort applied in person-months and  $KLOC$  is the estimated number of delivered lines of code for the project. The coefficient  $a_i$  and the exponent  $b_i$  are given in Table 2.

**Table 2 INTERMEDIATE COCOMO MODEL**

Software project	$a_i$	$b_i$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

The duration and person estimate is same as in basic COCOMO model i.e

$$D = cb(E) d_b \text{ months}$$

$$P = E/D \text{ persons}$$

### 3) Detailed COCOMO Model

The detailed model uses the same equations for estimation as the Intermediate Model. But detailed model can estimate the effort ( $E$ ), duration ( $D$ ) and persons ( $P$ ) of each of development phases, subsystems, modules.

Four Phases in detailed COCOMO model are

1. Requirements Planning and Product Design (RPD)
2. Detailed Design (DD)
3. Code & Unit Test (CUT)
4. Integrate & Test (TT)

Phases	Very Low	Low	Nominal	High	Very High
RPD	1.80	0.85	1.00	0.75	0.55
DD	1.35	0.85	1.00	0.90	0.75
CUT	1.35	0.85	1.00	0.90	0.75
IT	1.50	1.20	1.00	0.85	0.70

## ESTIMATION

### 3. Write short notes on Estimation

May: 05, 06, 07, Dec-07,10.

Many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. Software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk.

Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form  $d = f(v_i)$  where  $d$  is one of a number of estimated values (e.g., effort, cost, project duration) and  $v_i$  are selected independent parameters (e.g., estimated LOC or FP).

### DECOMPOSITION TECHNIQUES

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller problems.

The decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, we must understand the scope of the software to be built and generate an estimate of its “size.”

## 1. Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

- (1) The degree to which you have properly estimated the size of the product to be built;
- (2) The ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);
- (3) The degree to which the project plan reflects the abilities of the software team; and
- (4) The stability of product requirements and the environment that supports the software engineering effort.

If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP). Putnam and Myers suggest four different approaches to the sizing problem:

- “Fuzzy logic” sizing: To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.
- Function point sizing: The planner develops estimates of the information domain characteristics.
- Standard component sizing: Software is composed of a number of different “standard components” that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions.
- Change sizing: This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, and deleting code) of modifications that must be accomplished.

## 2 Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation:

- (1) As estimation variables to “size” each element of the software and
- (2) As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common.

LOC or FP (the estimation variable) is then estimated for each function. Function estimates are combined to produce an overall estimate for the entire project. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain,

and then the appropriate domain average for past productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate. Using historical data or (when all else fails) intuition,

Estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. A three-point or expected value can then be computed.

The expected value for the estimation variable (size)  $S$  can be computed as a weighted average of the optimistic ( $s_{opt}$ ), most likely ( $s_m$ ), and pessimistic ( $s_{pess}$ ) estimates.

### **An Example of LOC-Based Estimation**

Following the decomposition technique for LOC, an estimation table is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic, 4600 LOC; most likely, 6900 LOC; and pessimistic, 8600 LOC. Applying Equation the expected value for the 3D geometric analysis functions is 6800 LOC. Other estimates are derived in a similar fashion.

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	<i>33,200</i>

By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system. A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of \$8000 per month; the cost per line of code is approximately \$13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is \$431,000 and the estimated effort is 54 person-months.<sup>7</sup>

## An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the table we would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software. An FP value is computed using the technique

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	97
Number of external outputs	12	15	22	16	5	78
Number of external inquiries	16	22	28	22	5	88
Number of internal logical files	4	4	5	4	10	42
Number of external interface files	2	2	3	2	7	15
Count total						320

Each of the complexity weighting factors is estimated, and the value adjustment factor is computed as described in Chapter 23:

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
<b>Value adjustment factor</b>	<b>1.17</b>

FP estimated = count total × [0.65 + 0.01 × (F<sub>i</sub>)] = 375 The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of \$8000 per month, the cost per FP is approximately \$1230. Based on the FP estimate and the historical productivity data, the total estimated Project cost is \$461,000 and the estimated effort is 58 person-months.

Finally, the estimated number of FP is derived:

## 5. Process-Based Estimation

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function. Functions and related framework activities<sup>8</sup> may be represented as part of a table similar to the one presented. Once problem functions and process activities are melded, you estimate

the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table.

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity. It is very likely the labor rate will vary for each task. Senior staffs are heavily involved in early framework activities and are generally more expensive than junior staff involved in construction and release. Costs and effort for each function and framework activity are computed as the last step.

If process-based estimation is performed independently of LOC or FP estimation, we now have two or three estimates for cost and effort that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

## 6. Estimation with Use Cases

Developing an estimation approach with use cases is problematic for the following reasons [Smi99]:

- Use cases are described using many different formats and styles—there is no standard form.
- Use cases represent an external view (the user's view) of the software and can therefore be written at many different levels of abstraction.
- Use cases do not address the complexity of the functions and features that are described.
- Use cases can describe complex behaviors (e.g., interactions) that involve many functions and features.

Unlike an LOC or a function point, one person's "use case" may require months of effort while another person's use case may be implemented in a day or two.

LOC estimate =  $N \times \text{LOC}_{\text{avg}} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LOC}_{\text{adjust}}$

where

$N$  \_ actual number of use cases

$\text{LOC}_{\text{avg}}$  \_ historical average LOC per use case for this type of subsystem

$\text{LOC}_{\text{adjust}}$  \_ represents an adjustment based on  $n$  percent of  $\text{LOC}_{\text{avg}}$  where  $n$  is defined locally and represents the difference between this project and “average” projects

$S_a$  \_ actual scenarios per use case

$S_h$  \_ average scenarios per use case for this type of subsystem

$P_a$  \_ actual pages per use case

$P_h$  \_ average pages per use case for this type of subsystem

Expression could be used to develop a rough estimate of the number of LOC based on the actual number of use cases adjusted by the number of scenarios and the page length of the use cases. The adjustment represents up to  $n$  percent of the historical average LOC per use case.

3 c ) **Consider 7 functions with their estimated lines of code below. 8 marks**  
**[Nov / Dec 2016]**

Function	LOC
Func1	2340
Func2	5380
Func3	6800
Func4	3350
Func5	4950
Func6	2140
Func6	8400

**Average productivity based on historical data is 620 LOC/pm and labour rate is Rs.8000 per month. Find the total estimated project cost and effort.**

Solution:

Total estimation in LOC = 33360

A Review of historical data indicates :

1. Average productivity is 620 LOC per month
2. Average labour cost is \$8000 per month

Then cost for lines of code can be estimated as

Cost / LOC =  $8000 / 620 = \$12$

By considering total estimated LOC as 33360

Total estimated project cost =  $33360 \times 12 = \$400320$

Total estimated project effort =  $33360 / 620 = 54$  Person - months

## PROJECT SCHEDULING

### 4. Write short notes on i) Project Scheduling ii) Timeline Charts May: 05,06, 15

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. Schedule identifies all major process framework activities and the product functions to which they are applied. Scheduling for software engineering projects can be viewed from two rather different perspectives. I) first, an end date for release of a computer-based system. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization.

#### 1 Basic Principle

A number of basic principles guide software project scheduling: **Compartmentalization.** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.

**Interdependency.** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available.

**Time allocation.** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

**Effort validation.** Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time

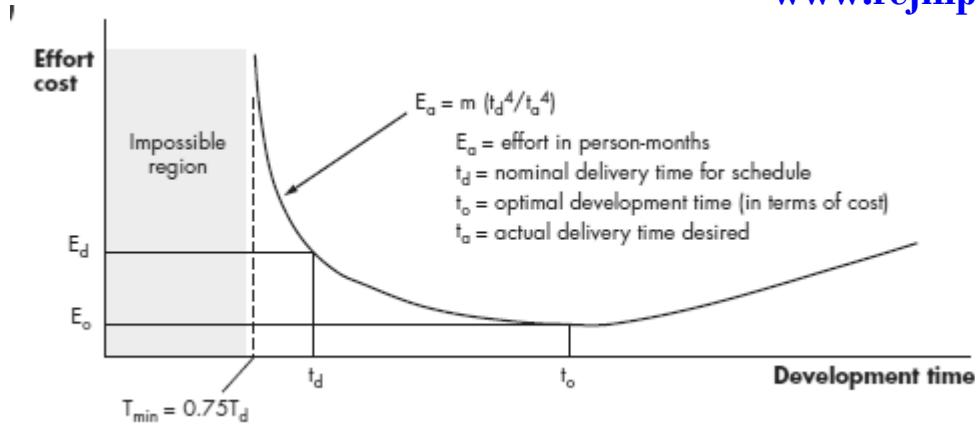
**Defined responsibilities.** Every task that is scheduled should be assigned to a specific team member.

**Defined outcomes.** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

**Defined milestones.** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

#### 2. The Relationship between People and Effort

In small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.



The curve indicates a minimum value  $t_o$  that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). The PNR curve also indicates that the lowest cost delivery option,  $t_o \approx 2t_d$ . The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay. The number of Delivered lines of code (source statements),  $L$ , is related to effort and development time by the equation:

$$L = P * E^{1/3} t^{4/3}$$

where  $E$  is development effort in person-months,  $P$  is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for  $P$  range between 2000 and 12,000), and  $t$  is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort  $E$ :

$$E = L^3 / P^3 t^4$$

Where  $E$  is the effort expended (in person-years) over the entire life cycle for software development and maintenance and  $t$  is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor (\$/person-year).

### Defining a task set for the software project

A task set is a collection of software engineering tasks, milestones and deliverables that must be accomplished to complete the project. A task network, called also activity network, is a graphic representation of the task flow of a project. It depicts the major software engineering tasks from the selected process model arranged sequentially or in parallel. Consider the task of developing a software library information system.

The scheduling of this system must account for the following requirements (the subtasks are given in *italics*): - initially the work should start with design of a control terminal ( $T_0$ ) class for no more than eleven working days; - next, the classes for student user ( $T_1$ ) and faculty user ( $T_2$ ) should be designed in parallel, assuming

that the elaboration of student user takes no more than six days, while the faculty user needs four days; -

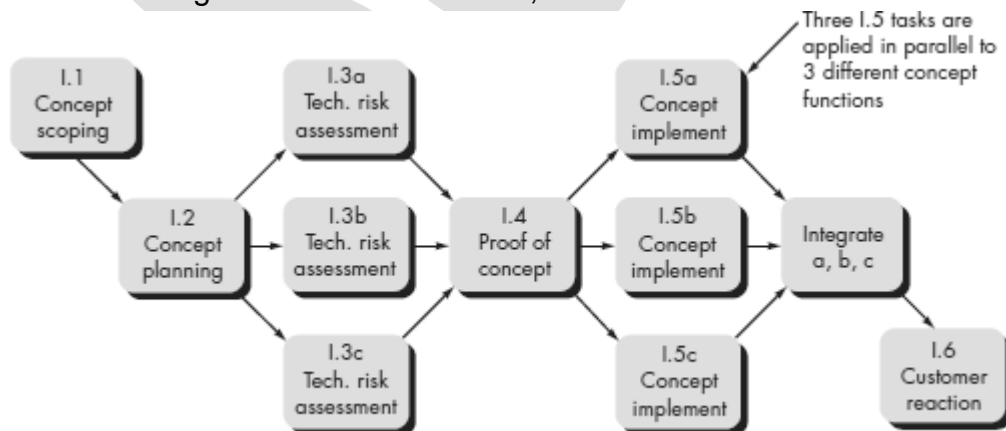
when the design of student user completes there have to be developed the network protocol (T4), it is a subtask that requires eleven days, and simultaneously there have to be designed network management routines (T5) for up to seven days; - after the termination of the faculty user subtask, a library directory (T3) should be made for nine days to maintain information about the different users and their addresses;

The completion of the network protocol and management routines should be followed by design of the overall network control (T7) procedures for up to eight days; - the library directory design should be followed by a subtask elaboration of library staff (T6), which takes eleven days; - the software engineering process terminates with testing (T8) for no more than four days

### Defining a task network

A task network, also called an activity network, is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering actions. A task network for concept development.

Program evaluation and review technique (PERT) and the critical path method (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected. Interdependencies among tasks may be defined using a task network. Tasks,



Sometimes called the project work breakdown structure (WBS), are defined for the product as a whole or for individual functions. Both PERT and CPM provide quantitative tools that allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish “most likely” time estimates

for individual tasks by applying statistical models, and (3) calculate “boundary times” that define a time “window” for a particular task.

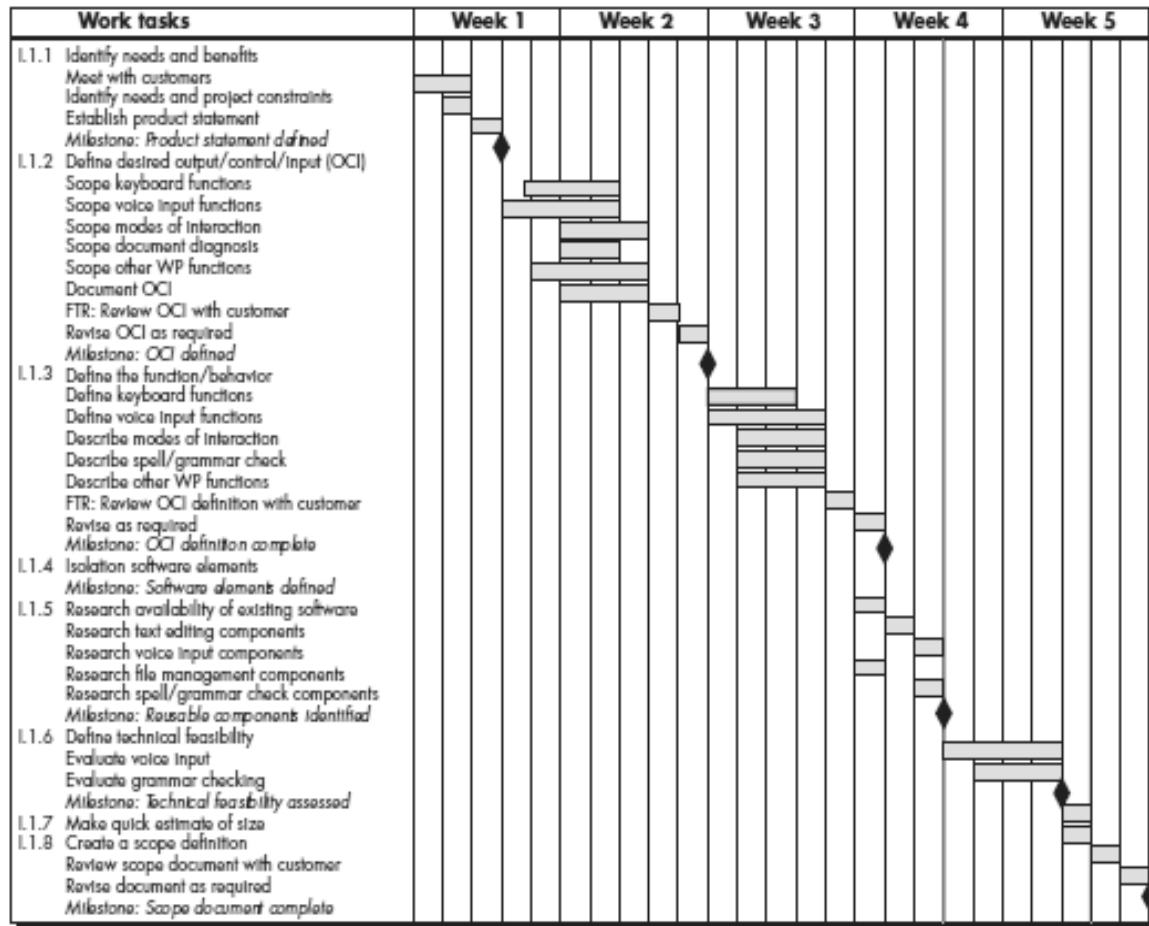
### **Time-line chart**

Time-line chart, also called a Gantt chart, is generated. A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual. Working on the project. Format of a time-line chart. It depicts a part of a software project schedule that emphasizes the concept scoping task for a word-processing (WP) software product. All project tasks are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones

### **Tracking the Schedule**

Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems
- Evaluating the results of all reviews conducted throughout the software engineering process
- Determining whether formal project milestones (the diamonds shown in Figure) have been accomplished by the scheduled date
- Comparing the actual start date to the planned start date for each project task listed in the resource table
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon



## Earned value analysis

EVA is a technique of performing quantitative analysis of the software project. It provides a common value scale for every task of software project.

The EVA acts as a measure for software project progress.

To determine the earned value, the following steps are performed:

1. The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWS<sub>i</sub> is the effort planned for work task i. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWS<sub>i</sub> values for all work tasks that should have been completed by that point in time on the project schedule.

2. The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence,

$$BAC = \sum (BCWS_k) \text{ for all tasks } k$$

3. Next, the value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Schedule performance index,  $SPI = BCWP / BCWS$

Schedule variance,  $SV = BCWP - BCWS$

Percent complete  $\% = BCWP / BAC$

## Effort Distribution

A recommended distribution of effort across the software process is often referred to as the *40–20–40 rule*. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk.

Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered. Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated even higher percentages are typical.

## RISK MANAGEMENT

### 5. What are the categories of software risks? Give an overview about risk management. May:14,15

Risk analysis and management are actions that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not.

Two characteristics of risk

- i) Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
- ii) Loss – the risk becomes a reality and unwanted consequences or losses occur

#### Different categories of risks are:

**i) 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

**ii) 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. Technical risks occur because the problem is harder to solve than you thought it would be.

**iii) 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). It is extremely important to note that simple risk categorization won't always work.

**iv) 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).

**v) 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).

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

### **Risk Identification**

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary. There are two distinct types of risks. Generic risks and Product-specific risks.

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

- Product size—risks associated with the overall size of the software to be built or modified.
- Business impact—risks associated with constraints imposed by management or the marketplace.
- Stakeholder characteristics—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
- Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization.

- Development environment—risks associated with the availability and quality of the tools to be used to build the product.
- Technology to be built—risks associated with the complexity of the system to be built and the “newness” of the technology that is packaged by the system.
- Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

## RISK PROJECTION

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

1. Establish a scale that reflects the perceived likelihood of a risk.
2. Delineate the consequences of the risk.
3. Estimate the impact of the risk on the project and the product.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings. Developing a Risk Table

A risk table provides you with a simple technique for risk projection.<sup>2</sup> A sample risk table is illustrated in Figure. We begin by listing all risks in the first column of the table. This can be accomplished with the help of the risk item checklists. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each

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

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

Risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of risk probability begins to converge. Next, the impact of each risk is assessed. Each risk component is assessed using the characterization presented 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. The cutoff line (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are reevaluated to accomplish second-order prioritization. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time.

### Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt.

The following steps to determine the overall consequences of a risk: (1) determine the average probability of occurrence value for each risk component; (2) determine the impact for each component based on the criteria and (3) complete the risk table and analyze the results. The overall risk exposure RE is determined using the following relationship  $RE = P \times C$ . Where  $P$  is the probability of occurrence for a risk, and  $C$  is the cost to the project should the risk occur.

For example, assume that the software team defines a project risk in the following manner:

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

**Risk probability.** 80 percent (likely).

**Risk impact.** Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is \$14.00, the overall cost (impact) to develop the components would be  $18 \times 100 \times 14 = \$25,200$ .

**Risk exposure.**  $RE = 0.80 \times 25,200 = \$20,200$ .

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

**PART - C**

1. Elaborate on business process engineering and product engineering. (NOV/DEC/2010)

OR

Business process engineering strives to define data and application architecture as well as technology infrastructure. Describe what each of these terms mean and provide an example. (NOV/DEC/2012)

OR

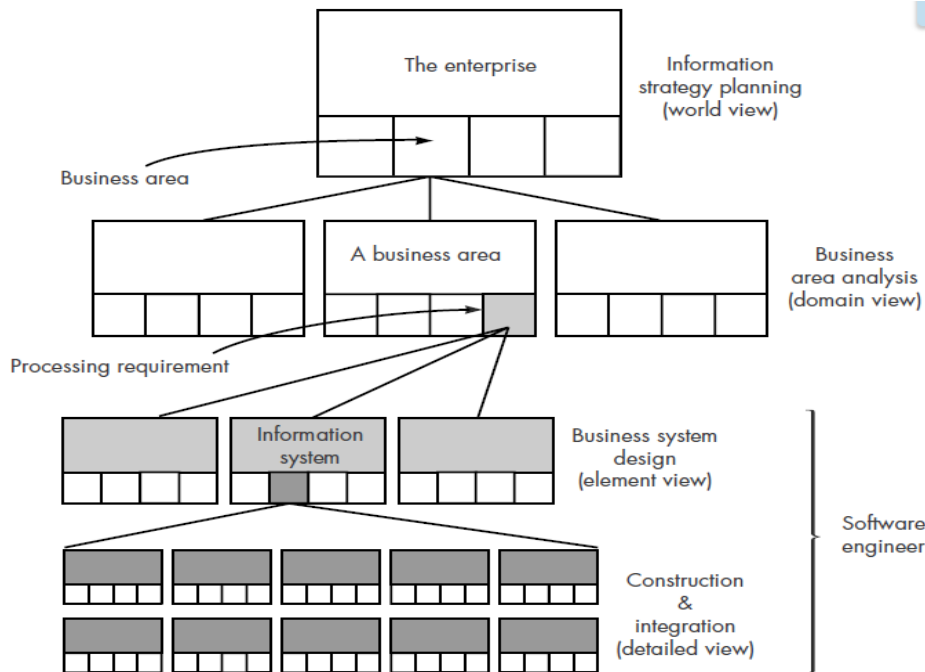
Explain the concept of business process engineering. (MAY/JUN/2012)  
(MAY/JUN/2013)

OR

Explain the concept of product engineering. (APR/MAY/2011)  
(MAY/JUN/2012)

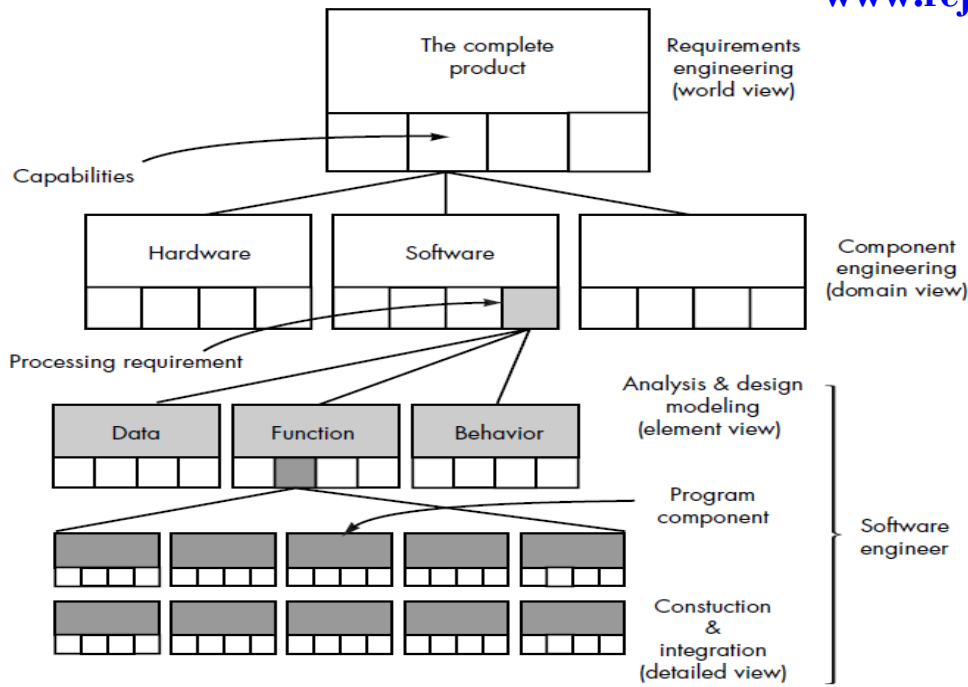
**Business process engineering Hierarchy**

- The goal of business process engineering (BPE) is to define architectures that will enable a business to use information effectively.
- Business process engineering is one approach for creating an overall plan for implementing the computing architecture.
- Three different architectures must be analyzed and designed within the context of business objectives and goals:
  - data architecture
  - applications architecture
  - technology infrastructure
- The data architecture provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used by the business. A data object contains a set of attributes that define some aspect, quality, characteristic, or descriptor of the data that are being described.
- The application architecture encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we consider the application architecture to be the system of programs (software) that performs this transformation. However, in a broader context, the application architecture might incorporate the role of people (who are information transformers and users) and business procedures that have not been automated.
- The technology infrastructure provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support the application and data. This includes computers, operating systems, networks, telecommunication links, storage technologies, and the architecture (e.g., client/server) that has been designed to implement these technologies.



## Product Engineering

- The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering—like business process engineering—must derive architecture and infrastructure.
- System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering.
- Once allocation has occurred, system component engineering commences. System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering. Each of these engineering disciplines takes a domain-specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of the role of requirements engineering is to establish the interfacing mechanisms that will enable this to happen.



The element view for product engineering is the engineering discipline itself applied to the allocated component. For software engineering, this means analysis and design modeling activities (covered in detail in later chapters) and construction and integration activities that encompass code generation, testing, and support steps. The analysis step models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

## 2. List the principles of agile software development. (Nov/Dec 2016)

1. Customer satisfaction by early and continuous delivery of valuable software
2. Welcome changing requirements, even in late development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Best architectures, requirements, and designs emerge from self-organizing teams

12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

**3. Explain the various phases of software development life cycle (SDLC) and identify deliverables at each phase.(May/Jun 2011)**

A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis

Stage 2: Defining Requirements

Stage 3: Designing the product architecture

Stage 4: Building or Developing the Product

Stage 5: Testing the Product

Stage 6: Deployment in the Market and Maintenance

**Requirement Gathering and analysis:** All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification doc.

**System Design:** The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture.

**Implementation:** With inputs from system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality which is referred to as Unit Testing.

**Integration and Testing:** All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

**Deployment of system:** Once the functional and non functional testing is done, the product is deployed in the customer environment or released into the market.

**Maintenance:** There are some issues which come up in the client environment. To fix those issues patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

**4. Do you feel spiral model is a good enough model to be followed in industry? Justify your answer with two reasons. (Nov/Dec 2007)**

The spiral model is similar to the incremental model, with more emphasis placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed. Each subsequent spirals builds on the baseline spiral.

**Advantages of Spiral model:**

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

**UNIT- II REQUIREMENTS ANALYSIS AND SPECIFICATION**

Software Requirements: Functional and Non-Functional, User requirements, System requirements, Software Requirements Document – Requirement Engineering Process: Feasibility Studies, Requirements elicitation and analysis, requirements validation, requirements management-Classical analysis: Structured system Analysis, Petri Nets- Data Dictionary.

**PART – A**

**1. What do you mean by functional & non functional requirements? May: 14**

Functional requirements should describe all the required functionality or system services. These requirements are heavily dependent upon the type of software, expected users and the type of system where the software is used.

The non functional requirements define system properties and constraints. Various properties of a system can be: Reliability, response time, storage requirements. And constraints of the system can be: Input and output device capability, system representation.

**2. An “SRS” is traceable comment. Dec: 05, 06**

SRS means Software Requirements Specification. There may be chances of having new requirement of the software. Traceability is concerned with relationship between requirements their sources and the system design. Thus change in requirement is manageable if SRS is traceable. The traceability requirement is the relationship between dependent requirements.

### 3. What is the purpose of feasibility study?

A feasibility study is a study is a made to decide whether or not the proposed system is worth while.

The focus of feasibility study is to check

1. If the system contributes to organizational objectives.
2. If the system can be engineered using current technology.
3. If the system is within the given budget.
4. If the system can be integrated with other useful systems.

### 4. Define elicitation.

May: 15

Requirement elicitation means discovery of all possible requirements. After identifying all possible requirements the analysis on these requirements can be done software engineers communicate the end users or customers in order to find out certain information such as: application domain, expected services from the system.

### 5. What is data dictionary? Where it is used in software engineering?

May: 13 Nov : 16

The data dictionary represents all of the names used in the system model. The descriptions of the entities, relationships and attributes are also included in data dictionary. The data dictionary is used during structured analysis for providing the detail information of data objects used in DFD.

### 6. What do requirement processes involve?

May: 12

Requirements engineering process involves elicitation, analysis, and validation.

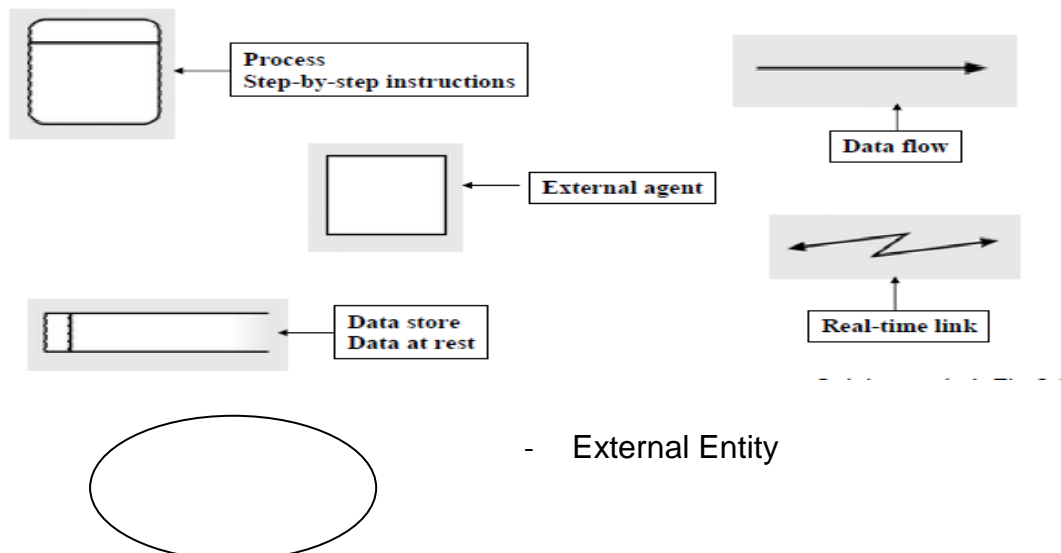
### 7. List two advantages of traceability tables in the requirements management phase. Dec :13

1. The traceability matrix ensures the coverage between different types of requirements.
2. The quick change impact analysis can be made using traceability matrix.

### 8. List the notations used in data flow models.

May: 14 ,16

## Data Flow Diagram Symbols



## 9. How the requirements are validated?

Requirement validation is a process in which it is checked that whether the gathered requirements represent the same system that customer really wants.

Requirement checking can be done in following manner: Validity, Consistency, Completeness, and Realism.

## 10. What is the major distinction between user requirements and system requirements?

May: 08

The user requirements describe both the functional and a non-functional requirement is such a way that they are understandable by the users who do not have detailed technical knowledge. These requirements are specified using natural languages tables and diagrams because these representations can be understood by all users. The system requirements are more detailed specifications of system functions, services and constraints than user requirements. These requirements can be expressed using system models.

## 11. What is prototype in software process? & what are the types of prototypes? OR

List any two advantages of prototypes. Dec: 13, May: 14, May :13.Nov :13

A prototype is an initial version of a system used to demonstrate concepts and try out design options.

A prototype can be used in:

- The requirements engineering process to help with requirements elicitation and validation;
- In design processes to explore options and develop a UI design; In the testing process to run back-to-back tests.

Types of prototypes

1. **Rapid prototyping** is a group of techniques used to quickly fabricate a scale model of a physical part or assembly using three-dimensional computer aided design (CAD) data.

2. **Evolutionary Prototyping** (also known as breadboard prototyping) is quite different from Throwaway Prototyping. The main goal when using Evolutionary Prototyping is to build a very robust prototype in a structured manner and constantly refine it.

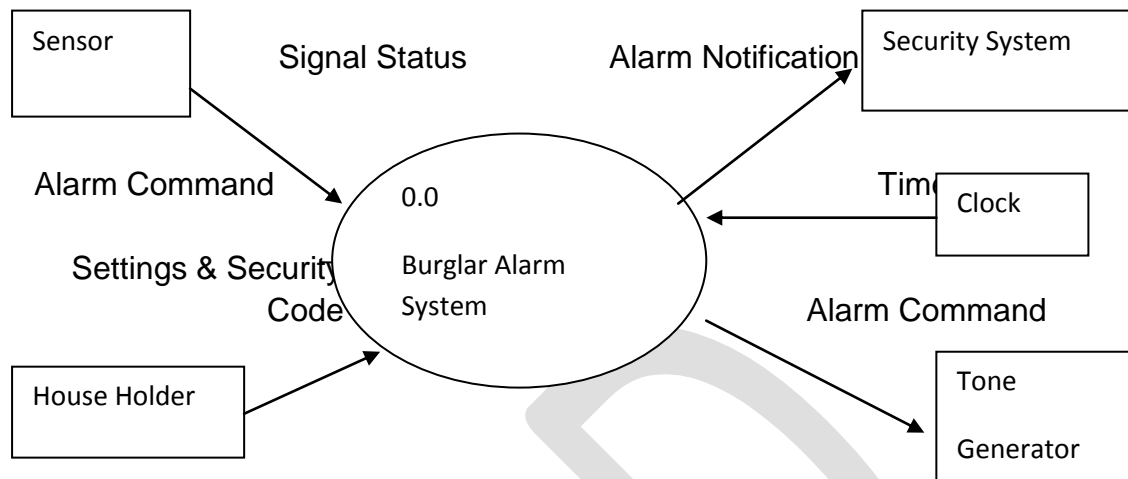
3. **Incremental prototyping** we can reduce the time gap between user and software developer.

4. **Extreme Prototyping** as a development process is used especially for developing web applications.

## 12. Define Requirement Management?

Requirement Management is the process of managing changing requirement during the requirement engineering process and system development.

13. Draw the context diagram for burglar alarm system.



14. List the good characteristics of a good SRS.

May : 16

- Correct
- Unambiguous
- Complete
- Consistent
- Specific
- Traceable

15. Classify the following as functional / non functional requirements for a banking system.

[Nov / Dec 2016]

- Verifying bank balance.
- Withdrawing money from bank.
- Completion of transactions in less than one second.
- Extending the system by providing more tellers for customers.

**Answer :**

Functional Requirements:

1. The system should allow the customer to verify the balance.
2. The system should dispense the cash on withdrawal.
3. The system should complete the transactions in less than one second.
4. The system should extend to customers by providing more tellers.

Non functional Requirements :

1. Each bank should process to verifying the bank balance for each customer.
2. The bank dispenses money only after the processing of withdrawal from the bank.
3. Each of the transaction should be made within one seconds. If the time limit is exceeded, then cancel the transaction automatically.

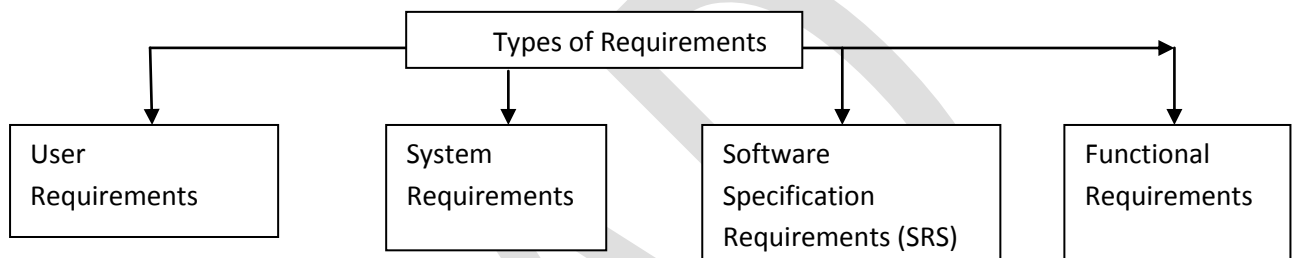
## PART-B

### FUNCTIONAL & NON-FUNCTIONAL REQUIREMENTS

#### 1. Explain functional & non-functional requirements.

May: 14,16

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



1. Functional Requirements

2. Non Functional Requirements

#### Types of requirements

##### User requirements

It is a collection of statements in natural language plus description of the service the system provides and its operational constraints. It is written for customers.

##### Guidelines for Writing User Requirements

##### For example

Consider a spell checking and correcting system a word processor. The user requirements can be given in natural language as the system should possess a traditional word dictionary and user supplied dictionary. It shall provide a user-activated facility which checks the spelling of words in the document against spellings in the system dictionary and user-supplied dictionaries.

When a word is found in the document which is not given in the dictionary, then the system should suggest 10 alternative words. These alternative words should be based on a match between the word found and corresponding words in the dictionaries. When a word is found in the document which is not in any dictionary, the system should propose following options to user:

1. Ignore the corresponding instance of the word and go to next sentence.
2. Ignore all instances of the word

3. Replace the word with a suggested word from the dictionary
4. Edit the word with user-supplied text
5. Ignore this instance and add the word to a specified dictionary

### **System Requirement**

System requirements are more detailed specifications of system functions, services and constraints than user requirements.

- They are intended to be a basis for designing the system.
- They may be incorporated into the system contract.
- The system requirements can be expressed using system models.
- The requirements specify what the system does and design specifies how it does.

System requirement should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. For a complex software system design it is necessary to give all the requirements in detail. Usually, natural language is used to write system requirements specification and user requirements.

### **Software specification**

It is a detailed software description that can serve as a basis for design or implementation. Typically it is written for software developers.

### **Functional Requirements**

Functional requirements should describe all the required functionality or system services.

The customer should provide statement of service. It should be clear how the system should react to particular inputs and how a particular system should behave in particular situation. Functional requirements are heavily dependent upon the type of software, expected users and the type of system where the software is used.

Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

For example: Consider a library system in which there is a single interface provided to multiple databases. These databases are collection of articles from different libraries. A user can search for, download and print these articles for a personal study.

From this example we can obtain functional Requirements as-

1. The user shall be able to search either all of the initial set of databases or select a subset from it.
2. The system shall provide appropriate viewers for the user to read documents in the document store.
3. A unique identifier (ORDER\_ID) should be allocated to every order. This identifier can be copied by the user to the account's permanent storage area.

### **Problems Associated with Requirements**

Requirements imprecision

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

### **Requirements completeness and consistency**

The requirements should be both complete and consistent. Complete means they should include descriptions of all facilities required. Consistent means there should be no conflicts or contradictions in the descriptions of the system facilities. Actually in practice, it is impossible to produce a complete and consistent requirements document.

### **Examples of functional requirements**

The LIBSYS system:

A library system that provides a single interface to a number of databases of articles in different libraries. Users can search for, download and print these articles for personal study. The user shall be able to search either all of the initial set of databases or select a subset from it.

The system shall provide appropriate viewers for the user to read documents in the document store. Every order shall be allocated a unique identifier (ORDER\_ID) which the user shall be able to copy to the accounts permanent storage area.

### **Non-Functional requirements**

Requirements that are not directly concerned with the specific functions delivered by the system Typically relate to the system as a whole rather than the individual system features Often could be deciding factor on the survival of the system (e.g. reliability, cost, response time)

### **Non-Functional requirements classifications:**

#### **Product requirements**

These requirements specify how a delivered product should behave in a particular way. Most NFRs are concerned with specifying constraints on the behavior of the executing system.

#### **Specifying product requirements**

Some product requirements can be formulated precisely, and thus easily quantified.

- Performance
- Capacity

Others are more difficult to quantify and, consequently, are often stated informally.

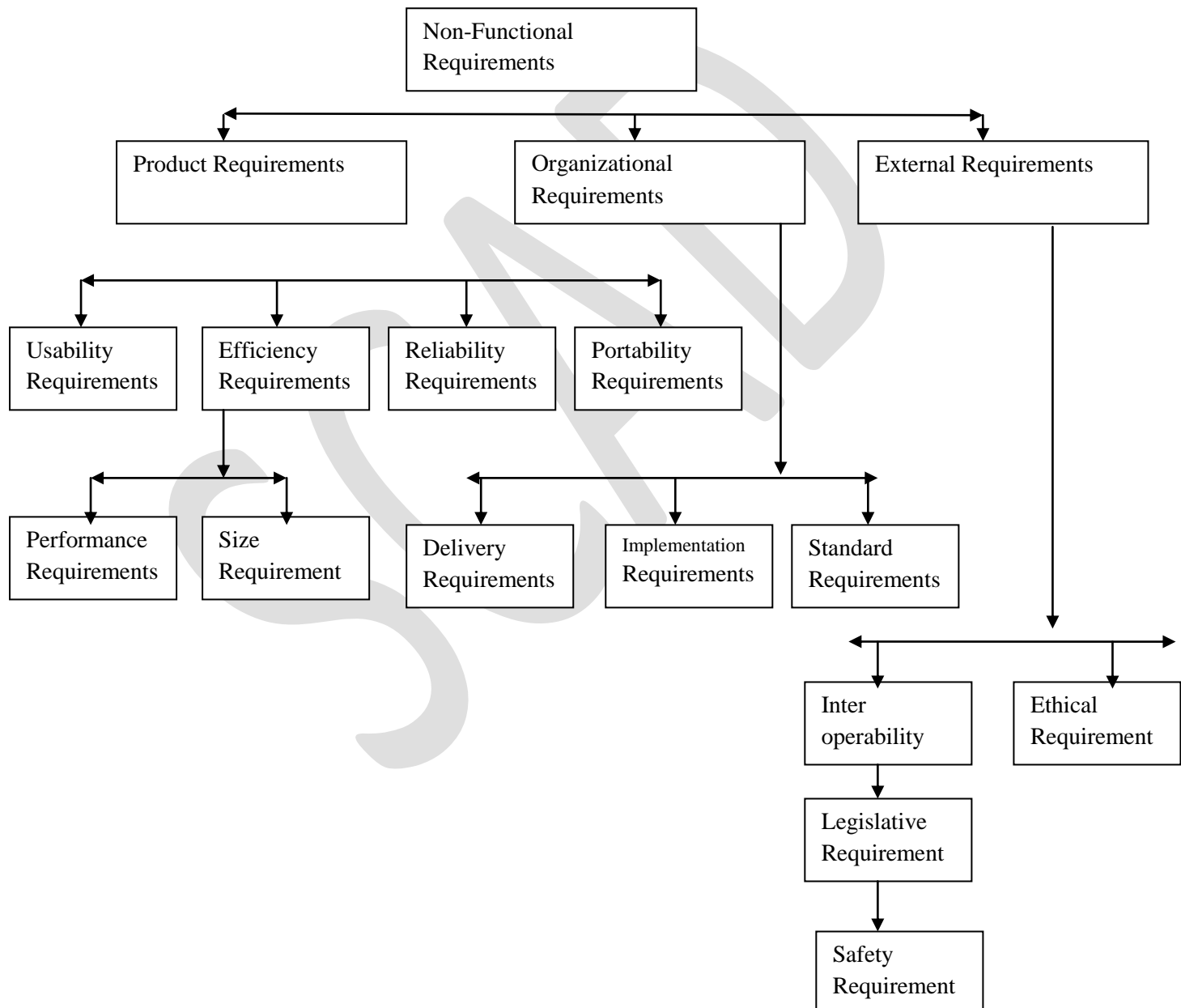
- Usability

#### **Process requirements**

Process requirements are constraints placed upon the development process of the system.

Process requirements include:

- Requirements on development standards and methods which must be followed
- CASE tools which should be used
- The management reports which must be provided



### **Examples of process requirements**

The development process to be used must be explicitly defined and must be conformant with ISO 9000 standards. The system must be developed using the XYZ suite of CASE tools.

Management reports setting out the effort expended on each identified system component must be produced every two weeks. A disaster recovery plan for the system development must be specified.

### **External requirements**

May be placed on both the product and the process. Derived from the environment in which the system is developed.

External requirements are based on:

- application domain information
- organizational considerations
- the need for the system to work with other systems
- health and safety or data protection regulations
- or even basic natural laws such as the laws of physics

### **Examples of external requirements**

Medical data system the organizations data protection officer must certify that all data is maintained according to data protection legislation before the system is put into operation.

Train protection system. The time required to bring the train to a complete halt is computed using the following function:

### **Organizational requirements**

The requirements which are consequences of organizational policies and procedures come under this category. For instance: process standards used implementation requirements.

## **SOFTWARE REQUIREMENT SYSTEM**

**2. Narrate the importance of SRS. Explain typical SRS structure and its parts. . Show the IEEE template of SRS document. Dec: 05, Nov: 12 ,May :16**

An SRS is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies at a particular point in time (usually) prior to any actual design or development work. It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

The SRS document itself states in precise and explicit language those functions and capabilities a software system (i.e., a software application, an eCommerce Web site, and so on) must provide, as well as states any required constraints by which the system must abide. The SRS also functions as a blueprint

for completing a project with as little cost growth as possible. The SRS is often referred to as the "parent" document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.

It's important to note that an SRS contains functional and nonfunctional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer's system requirements to be.

A well-designed, well-written SRS accomplishes four major goals:

It provides feedback to the customer. An SRS is the customer's assurance that the development organization understands the issues or problems to be solved and the software behavior necessary to address those problems. Therefore, the SRS should be written in natural language (versus a formal language, explained later in this article), in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables, and so on.

It decomposes the problem into component parts. The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps break down the problem into its component parts in an orderly fashion.

It serves as an input to the design specification. As mentioned previously, the SRS serves as the parent document to subsequent documents, such as the software design specification and statement of work. Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.

It serves as a product validation check. The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

SRSs are typically developed during the first stages of "Requirements Development," which is the initial product development phase in which information is gathered about what requirements are needed--and not. This information-gathering stage can include onsite visits, questionnaires, surveys, interviews, and perhaps a return-on-investment (ROI) analysis or needs analysis of the customer or client's current business environment. The actual specification, then, is written after the requirements have been gathered and analyzed.

SRS development process can offer several benefits:

Technical writers are skilled information gatherers, ideal for eliciting and articulating customer requirements. The presence of a technical writer on the requirements-gathering team helps balance the type and amount of information extracted from customers, which can help improve the SRS.

Technical writers can better assess and plan documentation projects and better meet customer document needs. Working on SRSs provides technical writers

with an opportunity for learning about customer needs firsthand--early in the product development process.

Technical writers know how to determine the questions that are of concern to the user or customer regarding ease of use and usability. Technical writers can then take that knowledge and apply it not only to the specification and documentation development, but also to user interface development, to help ensure the UI (User Interface) models the customer requirements.

Technical writers involved early and often in the process, can become an information resource throughout the process, rather than an information gatherer at the end of the process.

IEEE) have identified nine topics that must be addressed when designing and writing an SRS:

1. Interfaces
2. Functional Capabilities
3. Performance Levels
4. Data Structures/Elements
5. Safety
6. Reliability
7. Security/Privacy
8. Quality
9. Constraints and Limitations

An SRS document typically includes four ingredients are:

1. A template
2. A method for identifying requirements and linking sources
3. Business operation rules
4. A traceability matrix

**Table 1** A sample of a basic SRS outline

1.				Introduction
1.1				Purpose
1.2		Document		conventions
1.3		Intended		audience
1.4		Additional		information
1.5	Contact	information/SRS	team	members
1.6	References			
2.		Overall		Description
2.1		Product		perspective
2.2		Product		functions
2.3	User	classes	and	characteristics
2.4		Operating		environment
2.5		User		environment
2.6		Design/implementation		constraints

2.7	Assumptions and dependencies		
3.	External	Interface	Requirements
3.1		User	interfaces
3.2		Hardware	interfaces
3.3		Software	interfaces
3.4	Communication protocols and interfaces		
4.		System	Features
4.1	System	feature	A
4.1.1	Description	and	priority
4.1.2			Action/result
4.1.3		Functional	requirements
4.2	System feature B		
5.	Other	Nonfunctional	Requirements
5.1		Performance	requirements
5.2		Safety	requirements
5.3		Security	requirements
5.4	Software	quality	attributes
5.5		Project	documentation
5.6	User documentation		
6.		Other	Requirements
Appendix A:	Terminology/Glossary/Definitions		list
Appendix B:	To be determined		

**Table 2** A sample of a more detailed SRS outline

<b>1. Scope</b>	<p>1.1 Identification. Identify the system and the software to which this document applies, including, as applicable, identification number(s), title(s), abbreviation(s), version number(s), and release number(s).</p> <p>1.2 System overview. State the purpose of the system or subsystem to which this document applies.</p> <p>1.3 Document overview. Summarize the purpose and contents of this document. This document comprises six sections:</p> <ul style="list-style-type: none"> <li>• Scope</li> <li>• Referenced documents</li> <li>• Requirements</li> <li>• Qualification provisions</li> <li>• Requirements traceability</li> <li>• Notes</li> </ul>
-----------------	--

	Describe any security or privacy considerations associated with its use.
<b>2.Referenced Documents</b>	<p>2.1 Project documents. Identify the project management system documents here.</p> <p>2.2 Other documents.</p> <p>2.3 Precedence.</p> <p>2.4 Source of documents.</p>
<b>3. Requirements</b>	<p>This section shall be divided into paragraphs to specify the Computer Software Configuration Item (CSCI) requirements, that is, those characteristics of the CSCI that are conditions for its acceptance. CSCI requirements are software requirements generated to satisfy the system requirements allocated to this CSCI. Each requirement shall be assigned a project-unique identifier to support testing and traceability and shall be stated in such a way that an objective test can be defined for it.</p> <p>3.1 Required states and modes.</p> <p>3.2 CSCI capability requirements.</p> <p>3.3 CSCI external interface requirements.</p> <p>3.4 CSCI internal interface requirements.</p> <p>3.5 CSCI internal data requirements.</p> <p>3.6 Adaptation requirements.</p> <p>3.7 Safety requirements.</p> <p>3.8 Security and privacy requirements.</p> <p>3.9 CSCI environment requirements.</p> <p>3.10 Computer resource requirements.</p> <p>3.11 Software quality factors.</p> <p>3.12 Design and implementation constraints.</p> <p>3.13 Personnel requirements.</p> <p>3.14 Training-related requirements.</p> <p>3.15 Logistics-related requirements.</p> <p>3.16 Other requirements.</p> <p>3.17 Packaging requirements.</p> <p>3.18 Precedence and criticality requirements.</p>
<b>4.Qualification Provisions</b>	To be determined.

<b>5.Requirements Traceability</b>	To be determined.
<b>. Notes</b>	<p>This section contains information of a general or explanatory nature that may be helpful, but is not mandatory.</p> <p>6.1 Intended use. :This Software Requirements specification shall ...</p> <p>6.2 Definitions used in this document. Insert here an alphabetic list of definitions and their source if different from the declared sources specified in the "Documentation standard."</p> <p>6.3 Abbreviations used in this document. Insert here an alphabetic list of the abbreviations and acronyms if not identified in the declared sources specified in the "Documentation Standard."</p> <p>6.4 Changes from previous issue. Will be "not applicable" for the initial issue. Revisions shall identify the method used to identify changes from the previous issue.</p>

### **Establish a Requirements Traceability Matrix**

The business rules for contingencies and responsibilities can be defined explicitly within a Requirements Traceability Matrix (RTM), or contained in a separate document and referenced in the matrix, as the example in Table 3 illustrates. Such a practice leaves no doubt as to responsibilities and actions under certain conditions as they occur during the product-development phase.

The RTM functions as a sort of "chain of custody" document for requirements and can include pointers to links from requirements to sources, as well as pointers to business rules. For example, any given requirement must be traced back to a specified need, be it a use case, business essential, industry-recognized standard, or government regulation. As mentioned previously, linking requirements with sources minimizes or even eliminates the presence of spurious or frivolous requirements that lack any justification. The RTM is another record of mutual understanding, but also helps during the development phase.

## REQUIREMENTS ENGINEERING ACTIVITIES

**3. Write about the following Requirements Engineering activities. May: 15**

**i) Inception (2) ii) Elicitation (3) iii) Elaboration (3) iv) Negotiation (2)  
v) Specification (2) vi) Validation (2) vii) Requirements Management (2)**

Requirement engineering is a software engineering action that begins during communication activity and continues into the modeling activity.

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. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

### **1. Inception**

How does a software engineering 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?

At project inception, software engineers ask a set of questions that establish

- 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 the developer

### **2 Elicitations**

Christel and Kang identify a number of problems that help us understand the requirements elicitation is difficult:

- **Problems of scope.** The boundary of the system is ill-defined.
- **Problems of understanding.** The customers/users are not completely sure of what is needed, have poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or unstable.
- **Problems of volatility.** The requirements change over time.

### **3 Elaborations**

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 will interact with the systems.

## 4 Negotiations

It is not unusual for customers and users to ask for more than can be achieved given limited business resources. The requirements 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. Risks associated with each requirement are identified and analyzed; hence, rough “guest mates” of development efforts are made.

## 5. Specification

A specification can be any one (or more) of the following:

- A written document
- A set of models
- A formal mathematical
- A collection of usage scenarios (use-cases)
- A prototype

The specification is the final work product produced by the requirements engineer. It serves 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.

## 6 Validations

Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project and the product. The primary requirements 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.

# REQUIREMENT ENGINEERING PROCESS

**4. What is requirement engineering? Give the importance of feasibility study.**

**State its process and explain requirements elicitation problem. May:08,16**

## Requirement Engineering Process

The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification and requirements management .

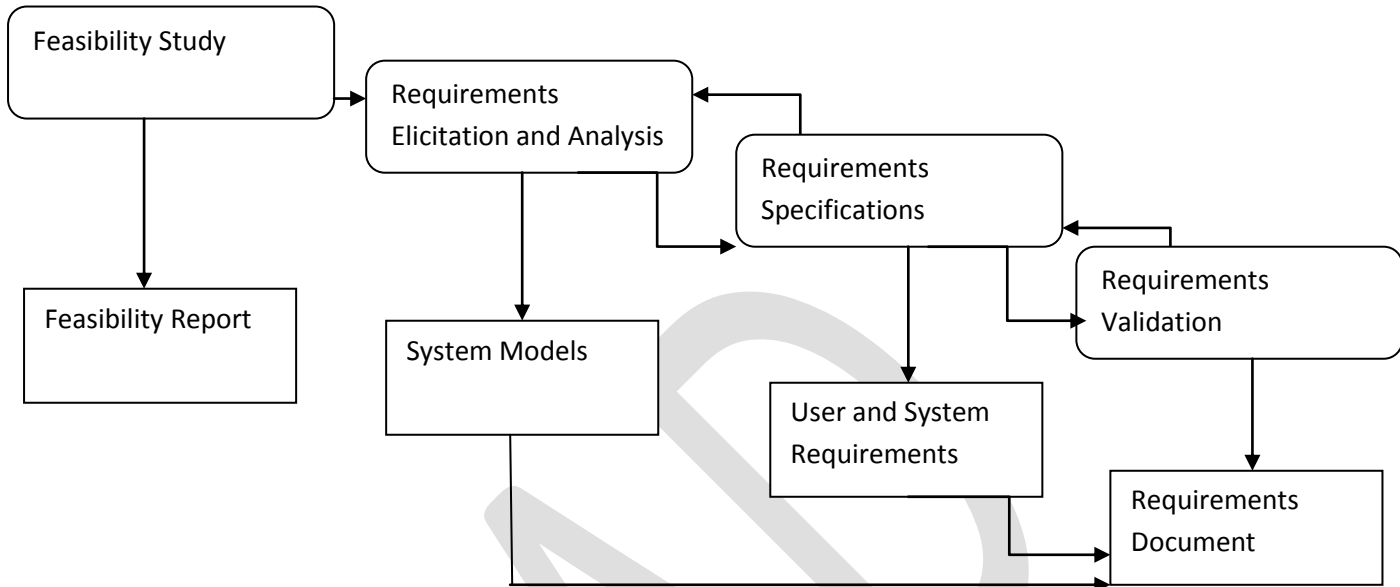
## Feasibility Studies

A feasibility study is a study made to decide whether or not the proposed system is worthwhile.

A short focused study that checks

- If the system contributes to organizational objectives

- If the system can be engineered using current technology and within budget
- If the system can be integrated with other systems that are used



- ☐ Based on information assessment (what is required), information collection and report writing

Questions for people in the organization

- What if the system wasn't implemented?
- What are current process problems?
- How will the proposed system help?
- What will be the integration problems?
- Is new technology needed? What skills?
- What facilities must be supported by the proposed system?

### Elicitation and analysis

[ Nov / Dec 2016 ]

It is sometimes called requirements elicitation or requirements discovery. It involves technical staff working with customers to find out about • The application domain • The services that the system should provide • The systems operational constraints

- ☐ May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions.

- These are called stakeholders

Problems of requirements analysis

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

- New stakeholders may emerge and the business environment change

### **System models**

- ☐ Different models may be produced during the requirements analysis activity
- ☐ Requirements analysis may involve three structuring activities which result in these different models
- Partitioning – Identifies the structural (part-of) relationships between entities
- Abstraction – Identifies generalities among entities
- Projection – Identifies different ways of looking at a problem
- ☐ System models will be covered on Scenarios
- ☐ Scenarios are descriptions of how a system is used in practice
- ☐ They are helpful in requirements elicitation as people can relate to these more readily than abstract statement of what they require from a system
- ☐ Scenarios are particularly useful for adding detail to an outline requirements description

### **Ethnography**

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

### **Requirements validation**

The purpose of requirements validation is to make sure that the customer and developer agree on details of the software requirements (or prototype) before beginning the major design work. This implies that both the customer and developer need to be present during the validation process. As each element of the analysis model is created, it is examined for consistency, omissions, and ambiguity.

The requirements represented by the model are prioritized by the customer and grouped within requirements packages that will be implemented as software increments and delivered to the customer.

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

Requirements validation techniques

- ☐ Reviews
  - Systematic manual analysis of the requirements
- ☐ Prototyping
  - Using an executable model of the system to check requirements.
- ☐ Testcase generation
  - Developing tests for requirements to check testability
- ☐ Automated consistency analysis
  - Checking the consistency of a structured requirements description

### **Requirements Management**

Requirements Management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.

### **ii) Initiating the Requirements Engineering Process**

The process of initiating engineering is the subject of this section. The point of view described is having all stakeholders (including customers and developers) involved in the creation of the system requirements documents.

The following are steps required to initiate requirements engineering:

#### **1. Identifying the Stakeholders**

A stakeholder is anyone who benefits in a direct or indirect way from the system which is being developed. Stakeholders are: operations managers, product managers, marketing people, internal and external customers, end-users, consultants, product engineers, software engineers, support and maintenance engineers, etc.

At inception, the requirements engineer should create a list of people who will contribute input as requirements are elicited. The initial list will grow as stakeholders are contacted because every stakeholder will be asked "Who else do you think I should talk to?"

#### **2 Recognizing Multiple Viewpoints**

Each of the stockholders will contribute to the RE process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

The requirements engineer is to categorize all stakeholder information including inconsistencies and conflicting requirements in a way that will allow decision makers to choose an internally inconsistent set of requirements for the system.

#### **3 Working toward Collaboration**

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (business manager, or a senior technologist) may make the final decision about which requirements make the final cut.

### **iii) Eliciting Requirements**

The requirements elicitation format that combines elements of problem solving, elaboration, and specification.

#### **1 Collaborative Requirements Gathering**

In this approach the stakeholders collaborate under the supervision of a neutral facilitator to determine the problem scope and define an initial set of requirements. Collaborative Requirements Gathering all apply some variation on the following guidelines:

- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) 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

**2 Quality Function Deployments (QFD)** In this technique the customer is asked to prioritize the requirements based on their relative value to the users of the proposed system. QFD is a technique that translates the needs of the customer into technical requirements for software engineering.

QFD identifies three types of requirements:

**Normal requirements:** These requirements reflect objectives and goals stated for a product or system during meetings with the customer.

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

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

In meeting with the customer, function deployment is used to determine the value of each function that is required for the system. Information deployment identifies both the data objects and events that the systems must consume and produce.

Finally, task deployment examines the behavior of the system within the context of its environment. And value analysis determines the relative priority of requirements determined during each of the three deployments.

#### **3 User Scenarios**

As requirements are gathered an overall version of system functions and features begins to materialize

Developers and users create a set of scenarios that identify a thread of usage for the system to be constructed in order for the software engineering team to understand how functions and features are to be used by end-users.

### **iv) Elicitation Work Products**

For most systems, the work product includes:

- 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 (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

#### **v) Developing Use-Cases**

Use-cases are defined from an actor's point of view. An actor is a role that users or devices play as they interact with the software engineer. The first case in writing a use-case is to define the set of "actors" that will be involved in the story. Actors are the different people or devices that use the system or product within the context of the functions and behavior that is to be described. An actor is anything that communicates with the system or product and that is external to the system itself. Because requirements elicitation is an evolutionary activity, not all actors are identified during the 1<sup>st</sup> iteration.

Primary actors interact to achieve required system functions and derive the intended benefit from the system. Secondary actors support the system so that primary action can do their work. Once actors have been identified, use-cases can be developed.

#### **vi) Building the Analysis Model**

The elements of the analysis model (scenario-based, class-based, behavioral, and flow-oriented) are introduced. The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dramatically as software engineers learn more about the system, and the stakeholders understand more about what they really require.

##### **1. Elements of the Analysis Model**

The specific elements of the analysis model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most analysis models.

**Scenario-based elements:** The system is described from the user's point of view using a scenario-based approach. It always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use-cases that describe how the software engineering models will be used. Functional—processing narratives for software functions.

**Class-based elements:** Each usage scenario implies a set of "objects" that are manipulated as an actor interacts with the system. These objects are categorized into classes- a collection of things that have similar attributes and common behaviors.

**Behavioral elements:** The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A state is any observable mode of behavior. Moreover, the state diagram indicates what actions are taken as a consequence of a particular event.

**Flow-oriented elements:** Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies functions to transform it; and produces output in a variety of forms.

### **vii) Negotiating Requirements**

The customer and the developer enter into a process of negotiation, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market.

The intent of the negotiations is to develop a project plan that meets the needs of the customer while at the same time reflecting the real-world constraints (time, people, and budget) that have been imposed on the software engineering team.

Boehm defines a set of negotiation activities at the beginning of each software process iteration. The following activities are defined:

- Identify the key stakeholders. These are the people who will be involved in the negotiation.
- Determine each of the stakeholders “win conditions”. Win conditions are not always obvious.
- Negotiate; work toward a set of requirements that lead to “win-win”

## **COMPONENTS OF SOFTWARE REQUIREMENTS**

### **5. What are the components of the standard structure for the software requirements document?**

**May: 14,16**

The data-flow approach is typified by the Structured Analysis method (SA)

Two major strategies dominate structured analysis

- Old method popularized by DeMarco
- Modern approach by Yourdon

#### **DeMarco**

- ☐ A topdown approach
- The analyst maps the current physical system onto the current logical data-flow model
- ☐ The approach can be summarized in four steps:
  - Analysis of current physical system
  - Derivation of logical model
  - Derivation of proposed logical model

- Implementation of new physical system

### **Modern structured analysis**

- ☐ Distinguishes between user's real needs and those requirements that represent the external behavior satisfying those needs
- ☐ Includes realtime extensions
- ☐ Other structured analysis approaches include:
  - Structured Analysis and Design Technique (SADT)
  - Structured Systems Analysis and Design Methodology (SSADM)

### **Method weaknesses**

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

### **CASE workbenches**

- ☐ A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.
- ☐ Analysis and design workbenches support system modelling during both requirements engineering and system design.
- ☐ These workbenches may support a specific design method or may provide support for creating several different types of system model.

### **Analysis workbench components**

- ☐ Diagram editors
- ☐ Repository and associated query language
- ☐ Report definition and generation tools
- ☐ Import/export translators
- ☐ Model analysis and checking tools
- ☐ Data dictionary
- ☐ Forms definition tools
- ☐ Code generation tools

### **Data Dictionary**

Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included

### **Advantages**

- Support name management and avoid duplication
  - Store of organisational knowledge linking analysis, design and implementation
- Many CASE workbenches support data dictionaries.

### **Petri Net :**

- ☐ Petri nets were invented by Carl Petri in 1966 to explore cause and effect relationships
- ☐ Expanded to include deterministic time
- ☐ Then stochastic time
- ☐ Then logic

### **Definition:**

- A Petri Nets (PN) comprises places, transitions, and arcs
  - Places are system states
  - Transitions describe events that may modify the system state
  - Arcs specify the relationship between places
- Tokens reside in places, and are used to specify the state of a PN

#### Data Dictionary Entries

Name	Description	Typed	Date
has-labels	1:N relation between entities of type Node or Link and entities of type Label	Relation	05.10.1998
Label	Holds structured or unstructured information about nodes or links Labels are represented by an icon.	Entity	08.12.1998
Link	A 1:1 relation between design entities represented as node, Links are types and may be named	Relation	08.12.1998
Name	Each label has a name which identifies the type of label the name must be unique within the set of label types used in a design.	Attribute	08.12.1998

EFTPOS example is a Petri net representation of a finite state machine (FSM).

It consists of three types of components: places (circles), transitions (rectangles) and arcs (arrows):

- Places represent possible states of the system;
- Transitions are events or actions which cause the change of state.

Every arc simply connects a place with a transition or a transition with a place is denoted by a movement of token(s) (black dots) from place(s) to place(s); and is caused by the firing of a transition. The firing represents an occurrence of the event or an action taken. The firing is subject to the input conditions, denoted by token availability.

- A transition is firable or enabled when there are sufficient tokens in its input places.
- After firing, tokens will be transferred from the input places (old state) to the output places, denoting the new state.

## CLASSICAL ANALYSIS

**5.What is classical analysis? Explain the performance of structured systems analysis and draw up Petri nets.**

- **What is the purpose of data flow diagrams? What are the notations used for the same.**

**Nov / Dec 2016**

Classical Analysis in software engineering and its allied technique, Structured Design (SD), are methods for analyzing and converting business requirements into

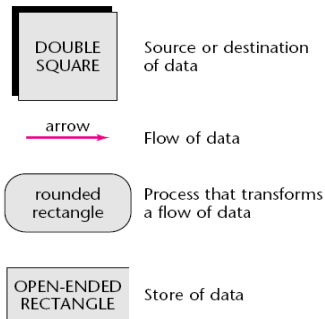
specifications and ultimately, computer programs, hardware configurations and related manual procedures.

- Specification document must satisfy two mutually contradictory requirements
- Must be clear and intelligible to client
  - Client probably not a computer expert
  - Client must understand it in order to authorize
- Must be complete and detailed
  - Sole source of information available to the design team
- Specification document must be in a format that is
  - Sufficiently nontechnical to be intelligible to client
  - Yet precise enough to result in a fault-free product
- Analysis (specification) techniques are needed
  - Classical (structured) analysis
  - Object-oriented analysis

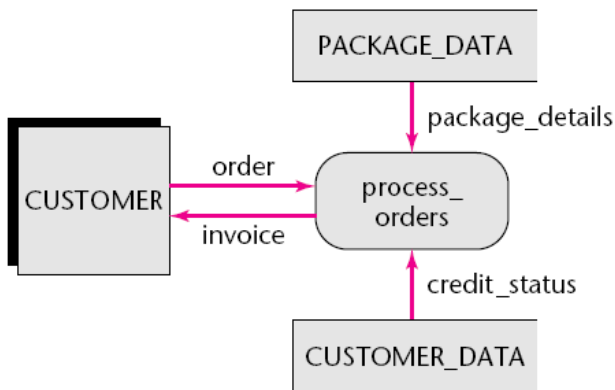
### **Structured Systems Analysis**

- Use of graphics to specify software
  - Important technique of the 1970s
  - A few popular techniques
- Structures system analysis — A nine-step technique to analyze client's needs
- Step-wise refinement is used in many of steps
- Step 1: Draw the Data Flow Diagram (DFD)
  - A pictorial representation of all aspects of the logical data flow
    - Logical data flow — what happens?
    - Physical data flow — how it happens
  - Any non-trivial product contains many elements
  - DFD is developed by stepwise refinement
  - For large products a hierarchy of DFDs instead of one DFD
  - Constructed by identifying data flows: Within requirements document or rapid prototype
  - Four basic symbols Source or destination of data (double-square box)
    - Data flow (arrow)
    - Process (rounded rectangle)
    - Data store (open-ended rectangle)
- Step 2: Decide what sections to computerize and how (batch or online)
  - Depending on client's needs and budget limitations
  - Cost-benefit analysis is applied

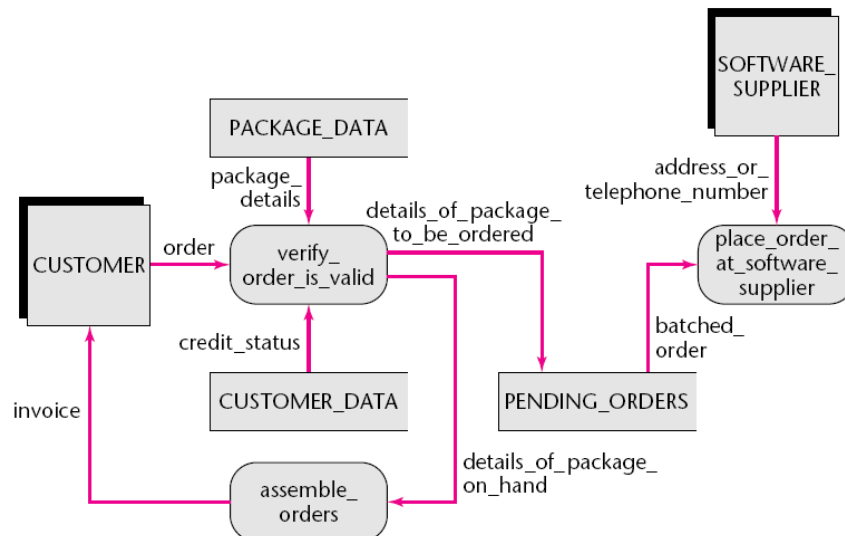
**The symbols of Gane and Sarsen's structured systems analysis.**



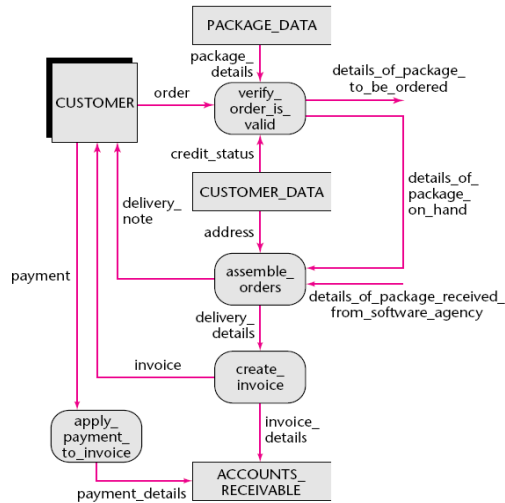
**The data flow diagram for Sally's Software Shop: first refinement.**



**The data flow diagram for Sally's Software Shop: second refinement.**



**The data flow diagram for Sally's Software Shop: part of third refinement.**



- Step 3: Determine details of data flows
  - Decide what data items must go into various data flows
  - Stepwise refinement of each flow
  - For larger products, a data dictionary is generated

Typical data dictionary entries for Sally's Software Shop.

Name of Data Element	Description	Narrative
order	Record comprising fields order_identification customer_details customer_name customer_address ... package_details package_name package_price ...	The fields contain all details of an order
order_identification	12-digit integer	Unique number generated by procedure generate_order_number. The first 10 digits contain the order number itself, the last 2 digits are check digits.
verify_order_is_valid	Procedure: Input parameter: order Output parameter: number_of_errors	This procedure takes order as input and checks the validity of every field; for each error found, an appropriate message is displayed on the screen (the total number of errors found is returned in parameter number_of_errors).

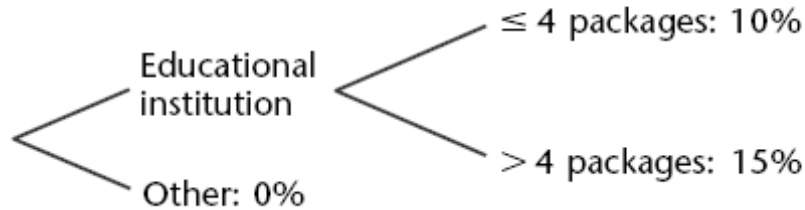
Step 4: Define logic of processes

- Determine what happens within each process
- Use of decision trees to consider all cases

**A decision tree depicting Sally's Software Shop educational discount policy.**

Give educational discount

- 



Step 5: Define data stores

- Exact contents of each store and its representation (format)

• Step 6: Define physical resources

- File names, organization (sequential, indexed, etc.), storage medium, and records
- If a database management system (DBMS) used: Relevant information for each table

• Step 7: Determine input-output specifications

- Input forms and screens
- Printed outputs

• Step 8: Determine sizing

- Computing numerical data to determine hardware requirements
- Volume of input (daily or hourly)
- Frequency of each printed report and its deadline
- Size and number of records of each type to pass between CPU and mass storage
- Size of each file

• Step 9: Determine hardware requirements

- Use of sizing information to determine mass storage requirements
- Mass storage for backup
- Determine if client's current hardware system is adequate

• After approval by client: Specification document is handed to design team, and software process continues

### Entity-Relationship Modeling

Entity-relationship modeling (ERM) — A semiformal data-oriented technique for specifying a product

- Emphasis is on data instead of operations
- Widely used for over 30 years specifying databases

### Finite State Machines

• Finite state Machines (FSMs) — A powerful formalism for specifying a product that can be modeled in terms of states and transitions between states

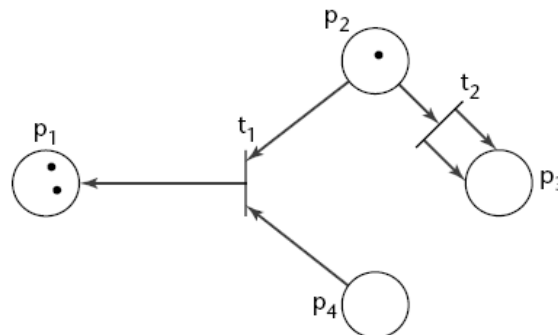
• Consists of five parts

- (1) A sets of states

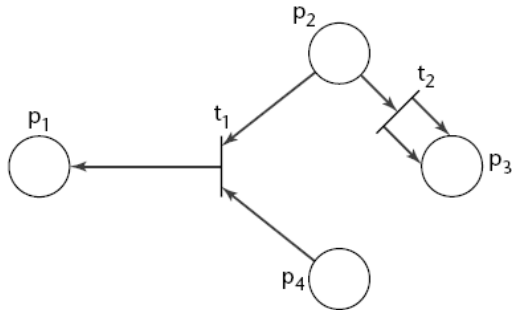
- (2) A set of inputs
- (3) Transition function: State transition diagram (STD)
- (4) Initial state
- (5) Set of final states
- Use of FSM widespread in computing applications
  - Every menu-driven user interface: an implementation of an FSM
  - Display of a menu: A state
  - Entering an input: An event, going to a new state

### Petri Nets

- Petri nets — Formal technique for describing concurrent interrelated activities
- Invented by Carl Adam Petri, 1962
- Consists of four parts
  - (1) A set of places
  - (2) A set of transitions
  - (3) An input function
  - (4) An output function
- Originally of interest to automata theorists
- Found wide applicability in computer science
- Performance evaluation
- Operating systems
- Software engineering
- Marking of a Petri net
- Assignment of tokens
- Tokens enable transitions
- Petri nets are non-deterministic
- If more than one transition is able to fire, then any one can be fired
- Use of inhibitor arcs an important extension:
  - Small circle instead of arrow
- A transition is enabled if
  - At least one token is in each of its (normal) input arcs and
  - No tokens on any of its inhibitor input arcs

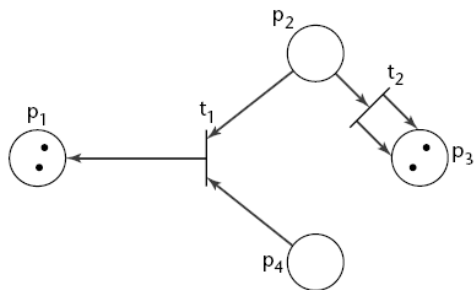


### A Petri net.

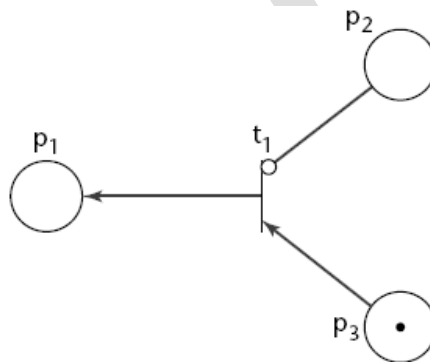


### A marked Petri net.

The Petri net after transition  $t_1$  fires.



### A Petri net with an inhibitor arc.



## PART -C

### 1) Sample problems related with Structured System Analysis.

1) Tamilnadu electricity Board (TNEB) would like to automate its billing process. Customers apply for a connection. EB staff take readings and update the system each customer is required to pay charges bi-monthly according to the rates set for the types of connection. Customers can choose to pay either by cash / card.

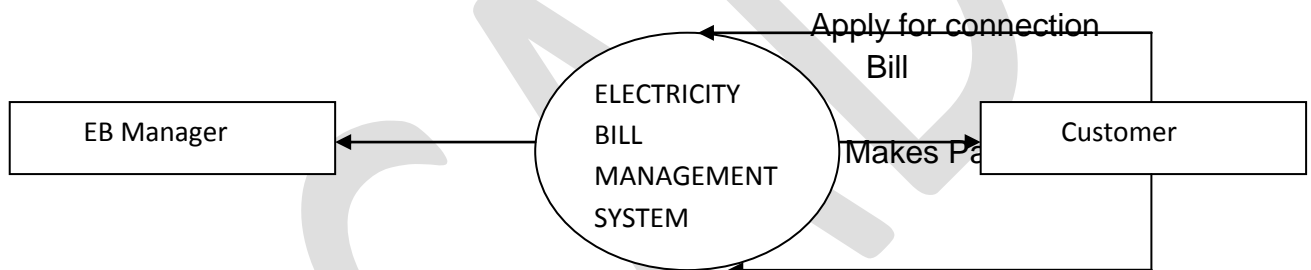
**Dec:13**

A bill is generated on payment

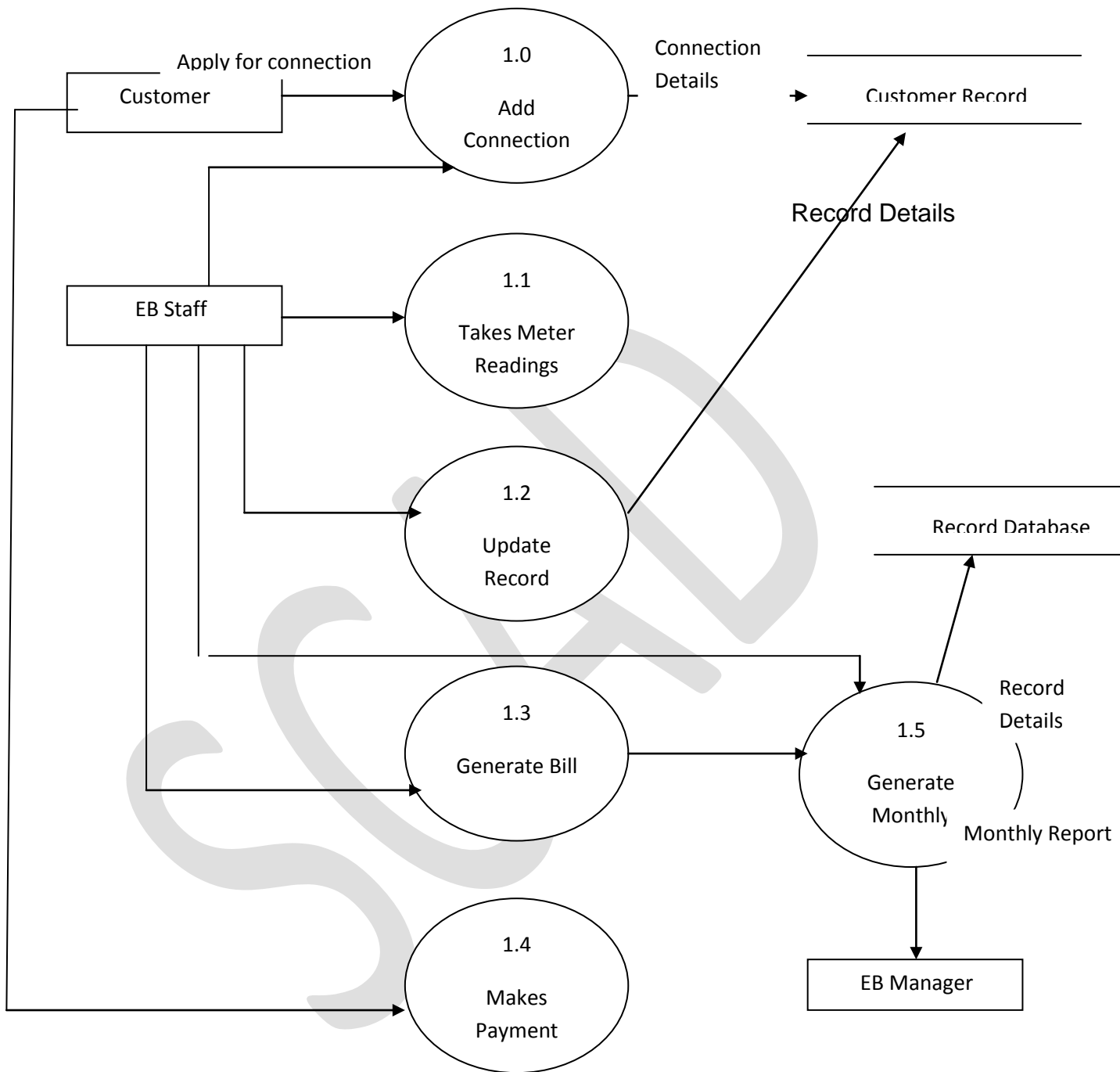
- i) Give a name for the system.
- ii) Draw the Level – 0 DFD
- iii) Draw the Level-1 DFD

**Ans : 1) Electricity Bill Management System**

**ii) Level – 0 DFD**



# Level- 1 DFD



2)A software is to be built that will control an Automated Teller Machine(ATM) . The ATM machine services customers 24 X 7. ATM has a magnetic stripe reader for reading an ATM card, a keyboard and display for interaction with the customer, a slot for depositing envelopes, a customer for cash, a printer for printing receipts and a switch that allows an operator to start / stop a machine.

The ATM services one customer at a time. When a customer inserts an ATM card and enters the personal identification number (PIN) the details are validated for each transaction. A customer can perform one or more transactions. Transactions made against each account are recorded so as to ensure validity of transactions.

If PIN is invalid, customer is required to re-enter PIN before making a transaction. If customer is unable to successfully enter PIN after three tries, card is retained by machine and customer has to contact bank.

The ATM provides the following services to the customer:

Withdraw cash in multiples of 100.

Deposit cash in multiples of 100

Transfer amount between any two accounts.

Make balance enquiry

Print receipt.

Dec:13

Ans:

1)

Stakeholders

Bank customer, Service operator, Hardware and software maintenance engineers, Database administrators, Banking regulators, Security administrator

Functional requirements

There should be the facility for the customer to insert a card.

The system should first validate card and PIN.

The system should allow the customer to deposit amount in the bank.

The system should dispense the cash on withdraw.

The system should provide the printout for the transaction.

The system should make the record of the transactions made by particular customer.

The cash withdrawal is allowed in multiple of 100

The cash deposition is allowed in multiple of 100.

The customer is allowed to transfer amount between the two accounts.

The customer is allowed to know the balance enquiry.

2. Non functional requirements.

Each of the transaction should be made within 60 seconds. If the time limit is exceeded , then cancel the transaction.

If there is no response from the bank computer after request is made within the minutes then the card is rejected with error message.

The bank dispenses money only after the processing of withdrawal form the bank. That means if sufficient fund is available in user's account then only the withdrawal request is processed.

Each bank should process the transactions from several ATM centres at the same time.

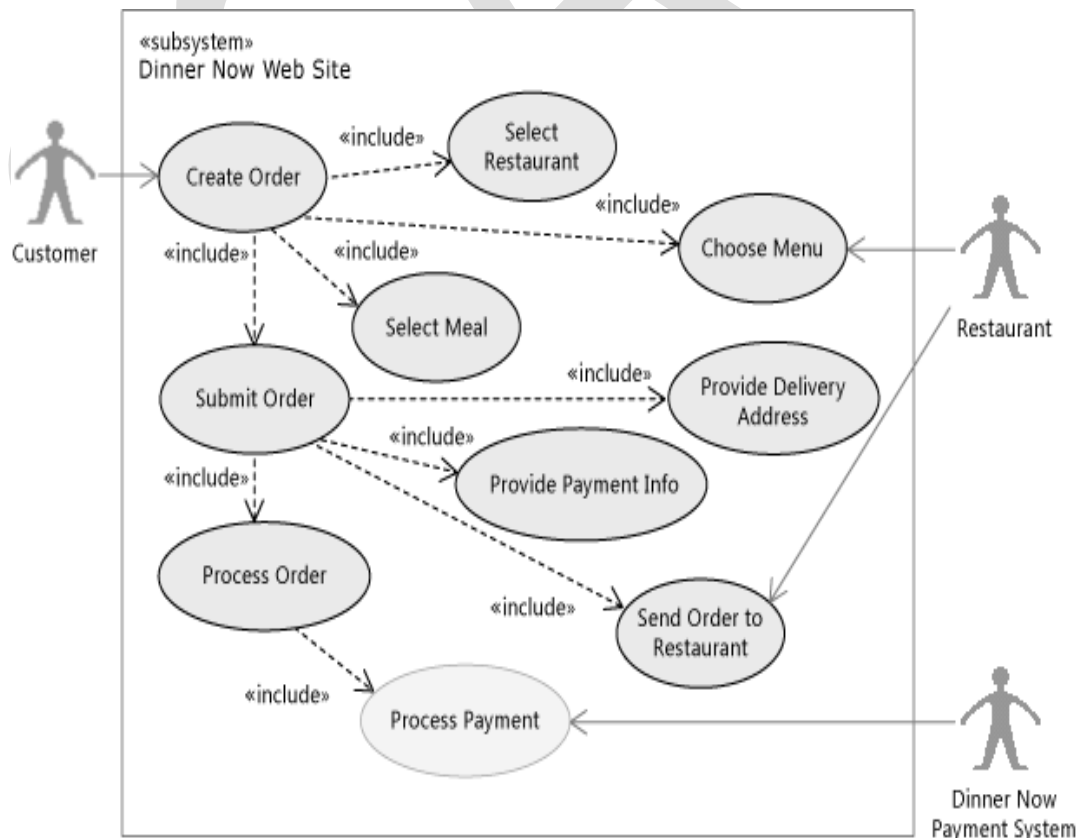
The machine should be loaded with sufficient fund in it.

**3) Draw Use Case and data Flow diagrams for a Restaurant System". The activates of the Restaurant system are listed below.**

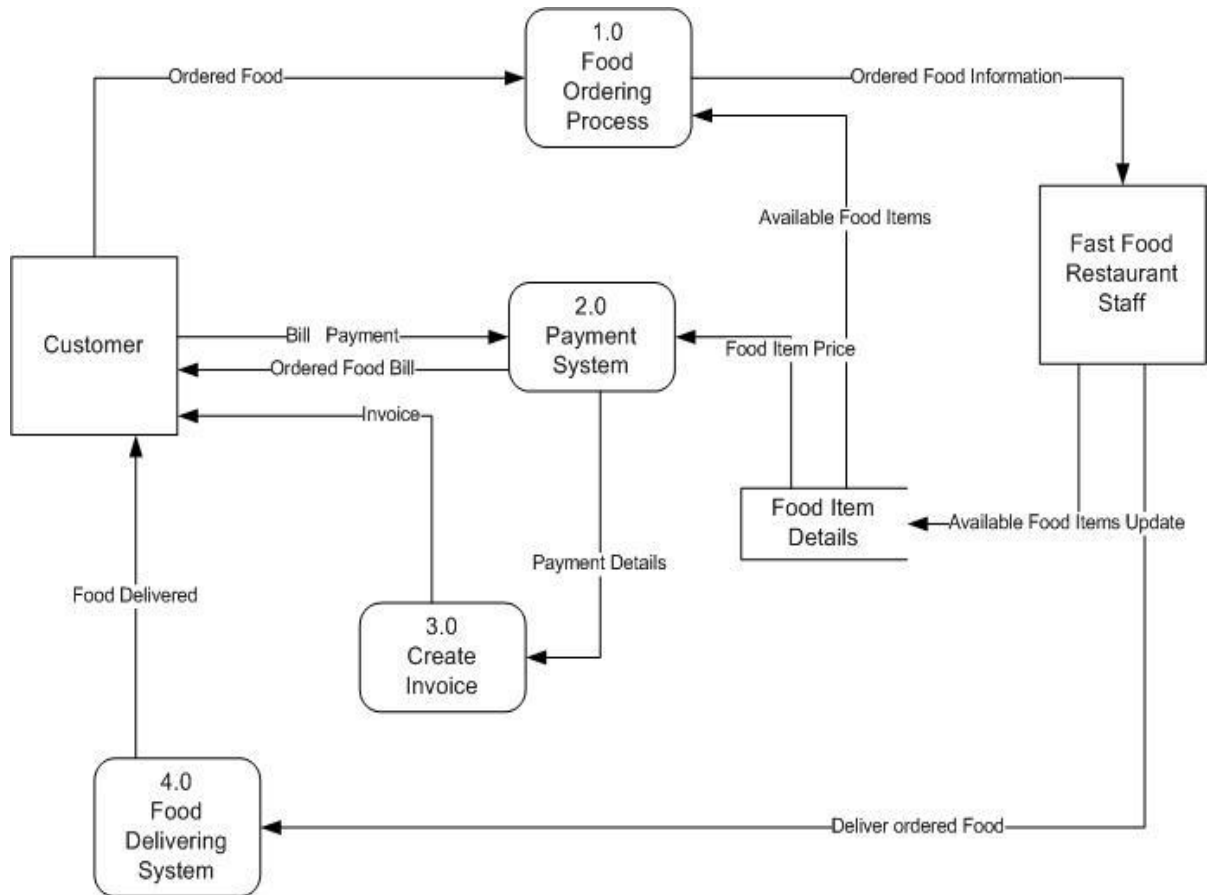
Receive the customer food orders , Produce the customer ordered foods, Serve the customer with their ordered foods , Collect Payment form customers, Store customers payment details , Order Raw Mateirals and pay for labor

**May:2015**

**Use Case Diagram**



## Data Flow Diagram



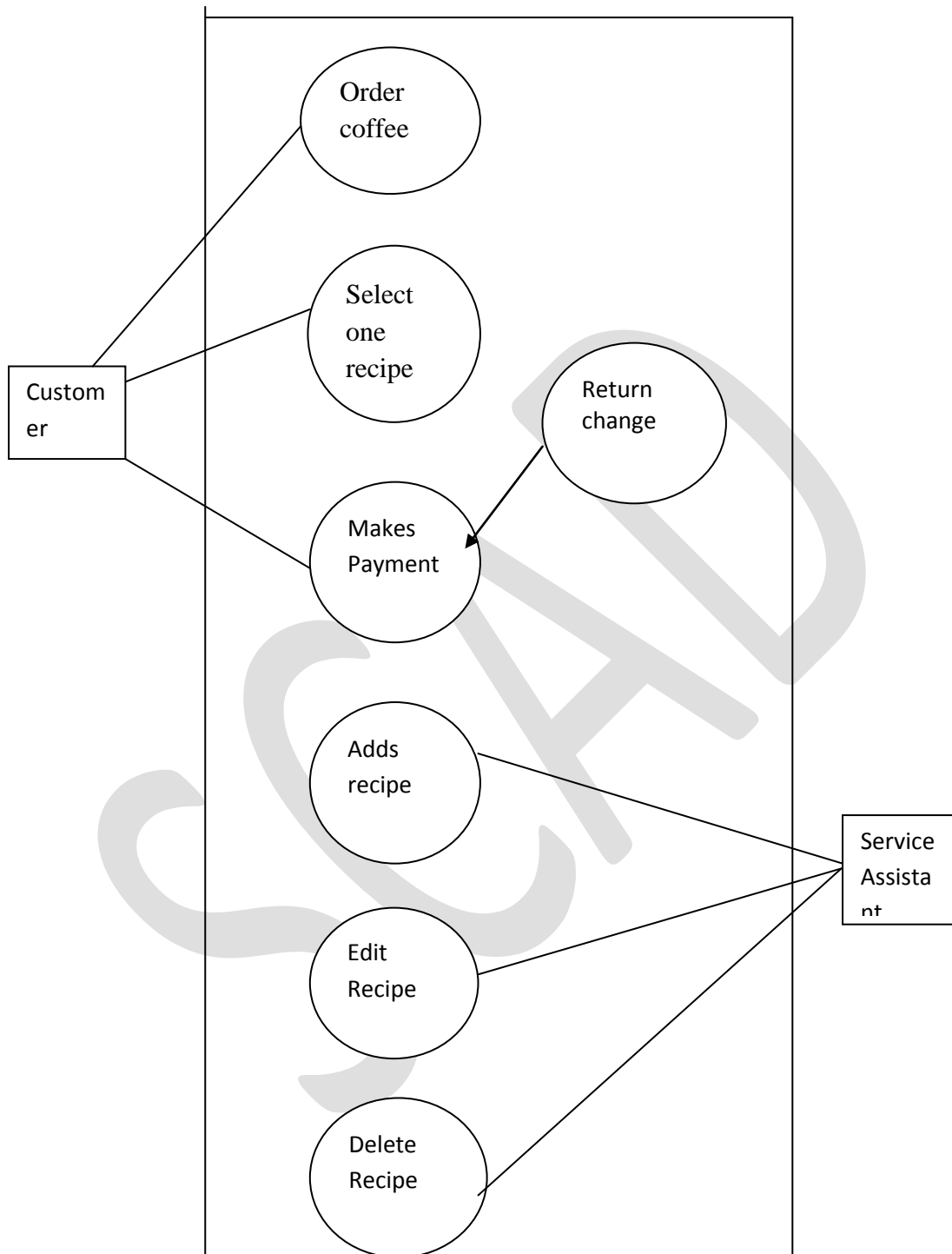
4. A coffee vending machine dispenses coffee to customers. Customers order coffee by selecting a recipe from a set of recipes. Customers pay for coffee using coins. Change is given back, if any, to the customer. The service assistant loads ingredients (coffee, Powder, milk, sugar, water, chocolate) into the coffee machine. The service assistant adds a recipe by water and chocolate to be added as well as the cost of the coffee.

The service assistant can also edit and delete a recipe.

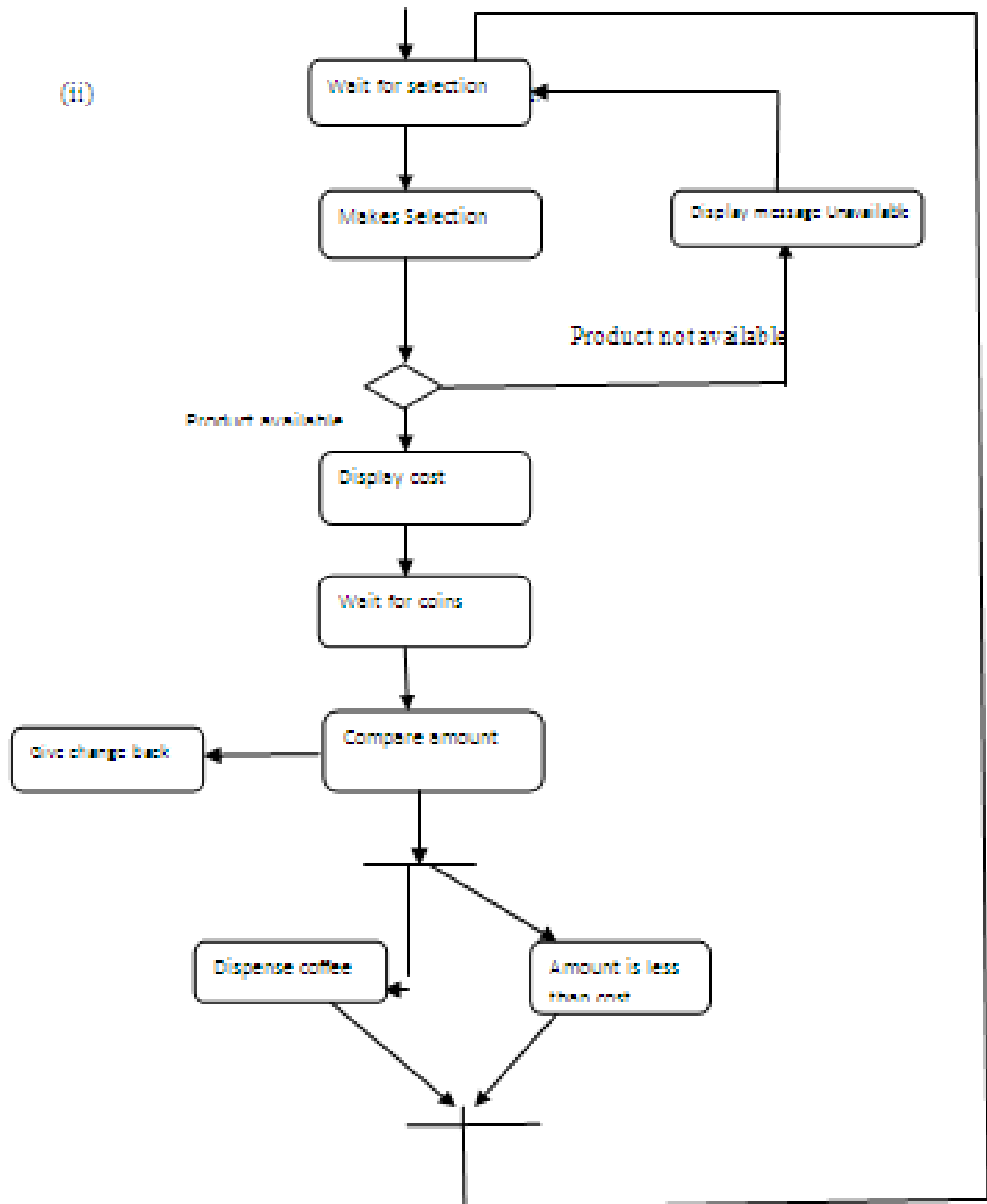
- (i) Develop the use case diagram for specification above
- (ii) For the scenario draw an activity diagram and sequence diagram.

DEC, 2013

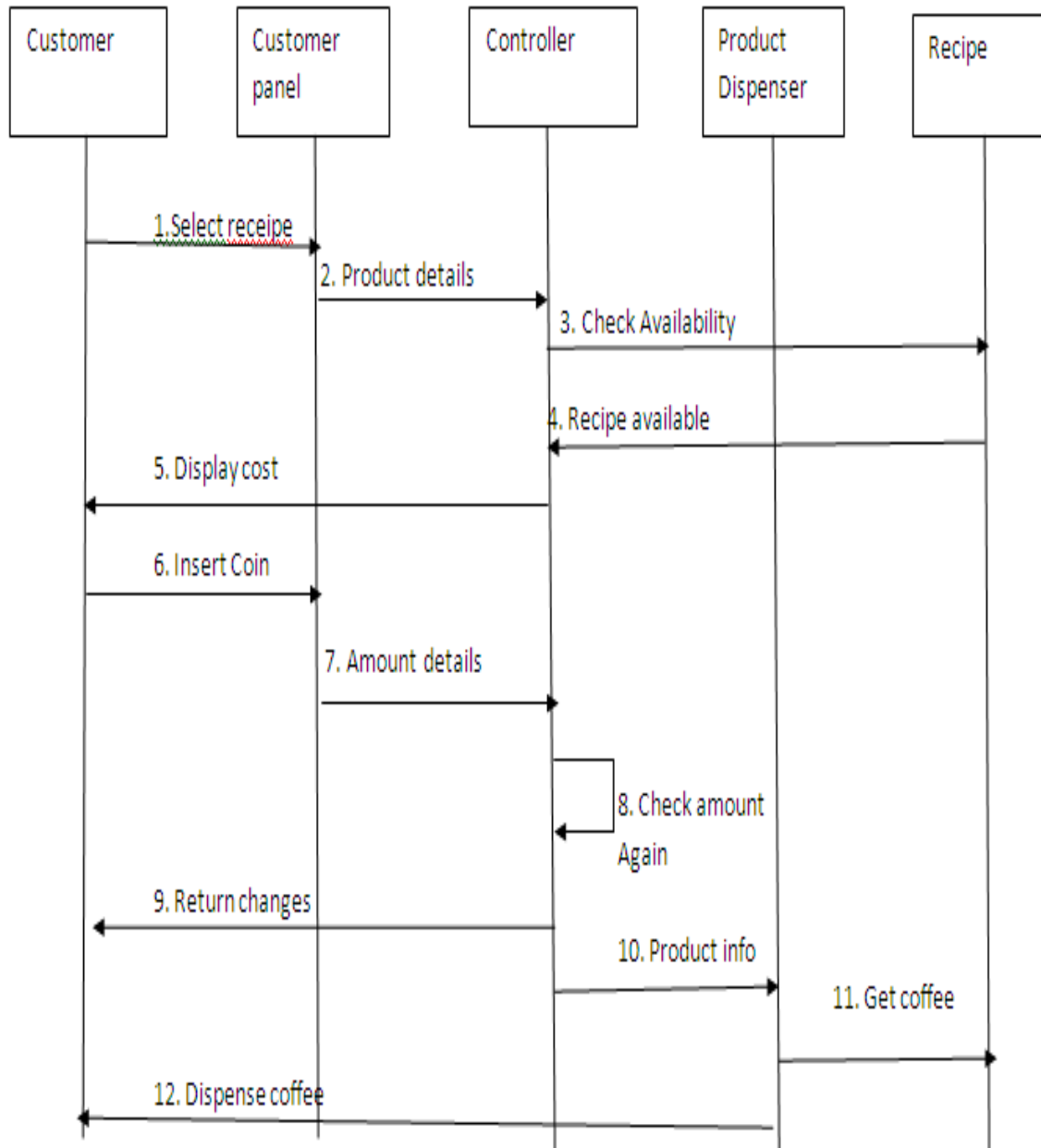
Use case diagram :



(ii)



## Sequence diagram



## UNIT III SOFTWARE DESIGN

Design process – Design Concepts-Design Model– Design Heuristic – Architectural Design –Architectural styles, Architectural Design, Architectural Mapping using Data Flow- User Interface Design: Interface analysis, Interface Design –Component level Design: Designing Class based components, traditional Components.

### PART – A

**1. List down the steps to be followed for user interface design. May: 15**

1. During the interface analysis step define interface objects and corresponding actions or operations.
2. Define the major events in the interface.
3. Analyze how the interface will look like from user's point of view.
4. Identify how the user understands the interface with the information provided along with it.

**2. Define software architecture May: 14**

Software architecture is a structure of systems which consists of various components, externally visible properties of these components and the inter-relationship among these components.

**3. Define abstraction. May: 13**

The abstraction means is a kind of representation of data in which the implementation details are hidden. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. A procedural abstraction refers to a sequence of instructions that have a specific and limited function.

**4. List out design methods. May: 12**

The two software design methods are

1. Object oriented design
2. Function oriented design

**5. What are the design qualities attributes 'FURPS' meant. Dec: 12**

FURPS stands for

1. Functionality: can be checked by assessing the set of features and capabilities of the functions.
2. Usability: Reliability is a measure of frequency and severity of failure.
3. Reliability: Is a measure of frequency and severity of failure.
4. Performance: It is a measure that represents the response of the system.
5. Supportability: It is the ability to adopt the changes made in the software.

6. List the difference between structured analyses and object oriented analysis. May: 08

Structured analysis	Object Oriented analysis
1. In Structured analysis the process and data are separately treated.	In Object Oriented analysis the process and data are encapsulated in the form of object.
2. It is not suitable for large and complex projects.	For large and complex system object oriented approach is more suitable.

7. List out four design principles of a good design

May: 09

1. The design process should not suffer from "tunnel vision".
2. The design should be traceable to analysis model.
3. The design should exhibit the uniformity and integrity.
4. Design is not coding and coding is not design.

8. What architectural styles are preferred for the following ? why ? Nov 2016

a) Net working

b) Web based Systems

C) Banking System

Net working – Data Centered Architecture. It posses the property of interchangeability.

Web based systems – Data flow architecture. It is the series of transformations are applied to produce the output data.

Banking system- Call & Retun architecture. In this style the hierarchical control fro call and return is represented.

9. What UI design patterns are used for the following?

[Nov / Dec 2016]

a) Page layout. b) Tables. C) Navigation through menus and web pages.

b) Shopping cart.

Answer :

Page layout – Page Grids

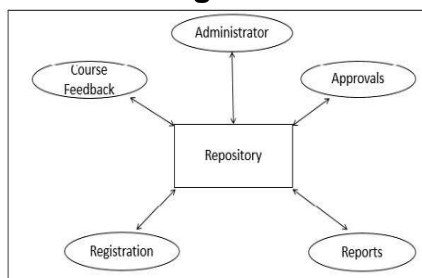
Navigation through menus and web pages- bread crump navigation.

Tables. – Form Wizard

Shopping cart – Information Dashboard

10. Draw diagram to demonstrate the architecture styles.

May: 15



11. 'A system must be loosely coupled and highly cohesive'. Justify.

Nov / Dec 2014

- coupling makes it possible to:
  - Understand one class without reading others
  - Change one class without affecting others
  - Thus: improves maintainability
- High cohesion makes it easier to:
  - Understand what a class or method does
  - Use descriptive names
  - Reuse classes or methods

12. If a module has logical cohesion, what kind of coupling is this module likely to have? May/June 2016

When a module that performs a tasks that are logically related with each other is called logically cohesive. For such module content coupling can be suitable for coupling with other modules.

The content coupling is a kind of coupling when one module makes use of data or control information maintained in other module.

13. Develop CRC model index card for a class account used in banking application. Nov / Dec 2013

Bank	Customer
Account Number	
Name	
Cash withdrawal	
Deposit	
Safety locker	

15. List two principles of good design.

Nov / Dec 2013

- Firmness : the program should not have any bugs that inhibit this function.
- Commodity : A program should be suitable for the purpose for which it was intended.
- Delight : the experience of using the program should be pleasurable.

16. What is the need for architectural mapping using data flow?

May/June 2016

A transform flow is a sequence of paths which forms transition in which input data are transformed into output data.

A transaction flow represents the information flow in which single data item triggers the overall information flow along the multiple paths.

**PART – B**  
**DESIGN PROCESS & DESIGN CONCEPTS**

**1. Explain about the various design process & design concepts considered during design.**  
**May: 03.06,07,08, Dec: 05**

**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. That is, the design is represented at a high level of abstraction a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.

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

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- 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.

**Some guidelines:**

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, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct 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**

- **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 (MTTF), the ability to recover from failure, and the predictability of the program.
- **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability

## Design Concepts

The software design concept provides a framework for implementing the right software. Following are certain issues that are considered while designing the software:

i) The **abstraction** means an ability to cope up with the complexity. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. A procedural abstraction refers to a sequence of instructions that have a specific and limited function.

An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.) A data abstraction is a named collection of data that describes a data object. 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, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction **door**.

ii) **Software architecture** means “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”. Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. 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 the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.

iii) **Patterns:** A pattern is a named nugget (something valuable) of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns". 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.

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

v) **Modularity** is the single attribute of software that allows a program to be intellectually manageable". Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

vi) **Information Hiding:** The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

vii) The concept of **functional independence** is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

Independence is assessed using two qualitative criteria: cohesion and coupling. Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

viii) **Refinement** is actually a process of elaboration. Refinement helps to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

ix) **Refactoring** is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or

inappropriate data structures, or any other design failure that can be corrected to yield a better design.

x) **Design classes** that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Five different types of design classes, each representing a different layer of the design architecture, can be developed:

- **User interface classes** define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- **Business domain classes** The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.
- **Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

## ARCHITECTURAL STYLES AND DESIGN

### 2. Explain in details about architectural styles and design. May: 07, 14

The architecture is not the operational software. Rather, it is a representation that enables Software Engineer to:

- (1) Analyze the effectiveness of the design in meeting its stated requirements,
- (2) Consider architectural alternatives at a stage when making design changes is still relatively easy, and
- (3) Reduce the risks associated with the construction of the software.

**Architectural model or style** is a pattern for creating the system architecture for given problem. Architectural style as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction.

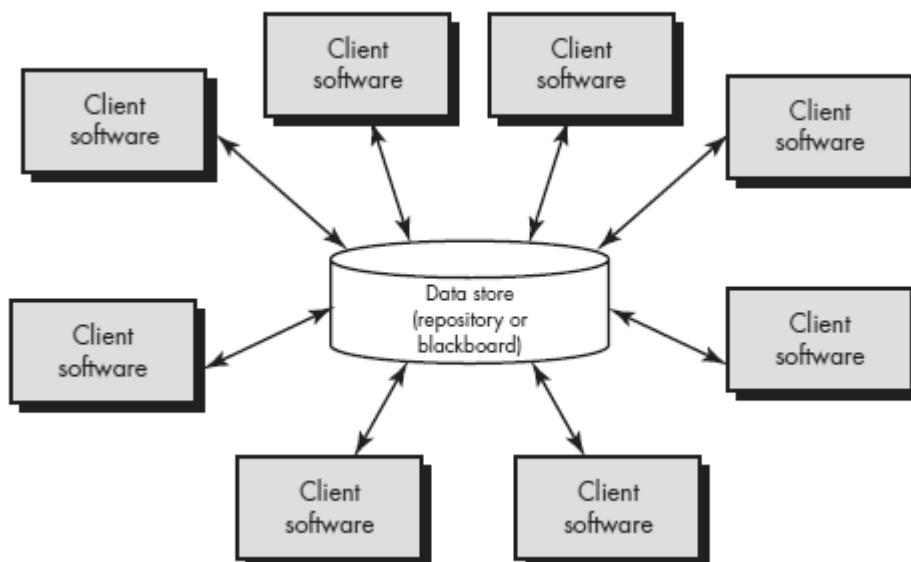
The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system; (2) a set of connectors that enable “communication,

coordination and cooperation” among components; (3) constraints that define how components can be integrated to form the system; and (4) semantic models that enable a designer to understand the overall properties of a system. An architectural style is a transformation that is imposed on the design of an entire system.

The intent is to establish a structure for all components of the system. An architectural pattern, like an architectural style, imposes a transformation on the design of architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety; (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)

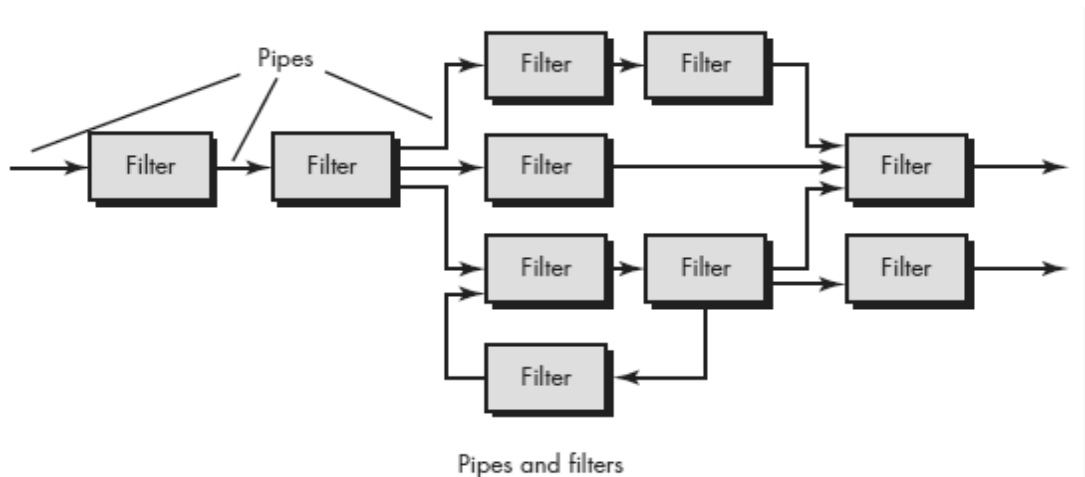
### **The commonly used architectural styles are**

**1. Data-centered architectures.** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes. Data-centered architectures promote inerrability. That is, existing



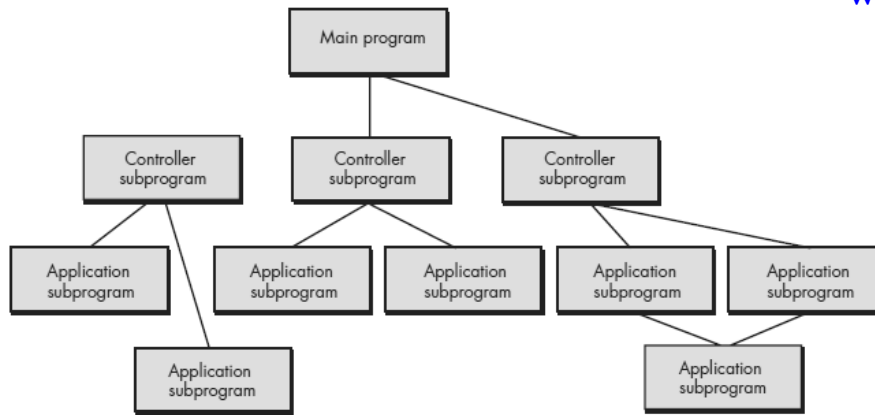
Components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

**2. Data-flow architectures.** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one



component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters. If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

**3. Call and return architectures.** This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:



- **Main program/subprogram architectures.** This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components. Figure illustrates architecture of this type.

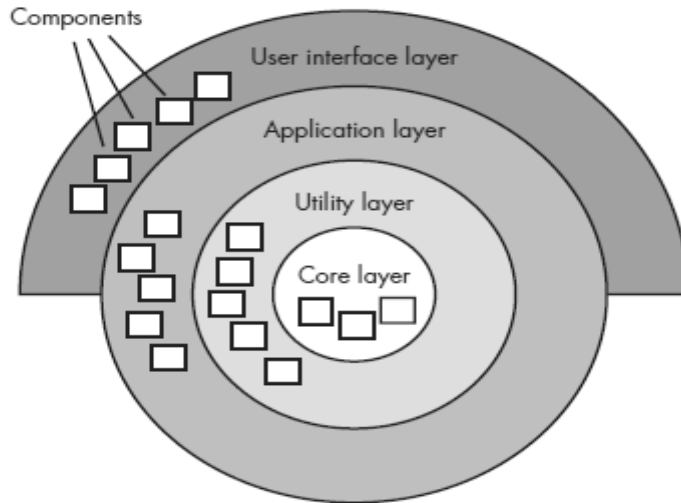
- **Remote procedure call architectures.** The components of a main program/subprogram architecture are distributed across multiple computers on a network.

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

**5. Layered architectures.** The basic structure of a layered architecture is illustrated in Figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

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

1. The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.
2. A pattern imposes a rule on the architecture, describing how the S/W will handle some aspect of its functionality at the infrastructure level.
3. Architectural patterns tend to address specific behavioral issues within the context of the architectural.



..

Fig: Layered Architecture

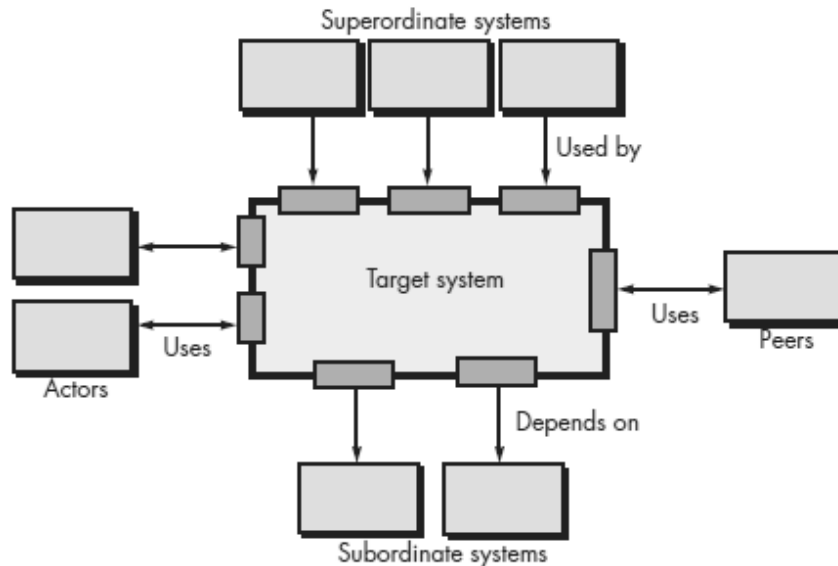
As **architectural design** begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An archetype is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

### Representing the System in Context

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

Referring to the figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

- Super ordinate systems—those systems that use the target system as part of some higher-level processing scheme.
- Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- Peer-level systems—those systems that interact on a peer-to-peer basis (i.e. information is either produced or consumed by the peers and the target system).
- Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.



## Defining Archetypes

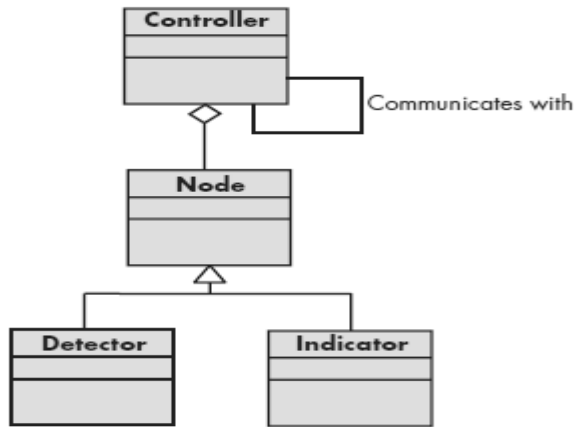
An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

Continuing the discussion of the Safe Home security function, we might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

**Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another. Each of these archetypes is depicted using UML notation as shown in Figure. For example, **Detector** might be refined into a class hierarchy of sensors.



The design analysis activities that follow are performed iteratively:

1. Collect scenarios: A set of use cases is developed to represent the system from the user's point of view.
2. Elicit requirements, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.
4. Evaluate quality attributes by considering each attribute in isolation. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.

## TRANSFORM AND TRANSACTIONAL MAPPING

3. **Describe transform and transactional mapping by applying design steps to an example system.** May: 06, Dec: 07, 08, 09
  - **What is structured design ? Illustrate the SD process from DFD to structured chart with a case study.** (Nov /Dec 2016)

A mapping technique, called structured design, is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram) to software architecture. The transition from information flow

(represented as a DFD) to program structure is accomplished as part of a six steps process:

- (1) The type of information flow is established,
- (2) Flow boundaries are indicated,
- (3) The DFD is mapped into the program structure,
- (4) Control hierarchy is defined,
- (5) The resultant structure is refined using design measures and heuristics, and
- (6) The architectural description is refined and elaborated.

“Transform” mapping for a small part of the Safe Home security function. In order to perform the mapping, the type of information flow must be determined. One type of information flow is called transform flow and exhibits a linear quality. Data flows into the system along an incoming flow path where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a transform center. Finally, it flows out of the system along an outgoing flow path that transforms the data into external world form.

#### Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To illustrate this approach, we again consider the SafeHome security function. One element of the analysis model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into software architecture, we would initiate the following design steps:

**Step 1. Review the fundamental system model.** The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 1 depicts a level 0 context models, and Figure 2 shows refined data flow for the security function.

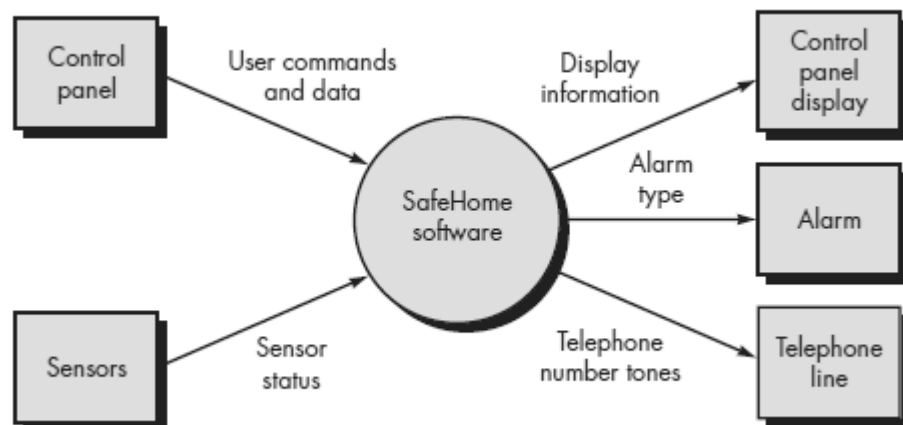


Fig:1 Context-level DFD for the SafeHome security function

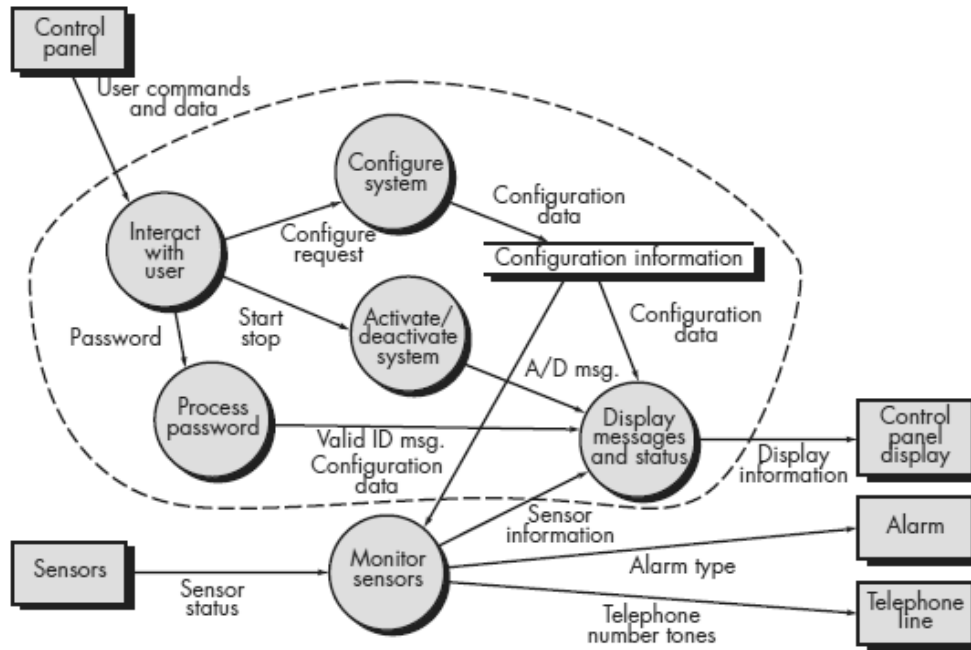


Fig:2 Level 1 DFD for the Safe Home security function Level 1 DFD for the Safe Home security

**Step 2. Review and refine data flow diagrams for the software.** Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors (Figure 3) is examined, and a level 3 data flow diagram is derived as shown in Figure 4. At level 3, each transform in the data flow diagram exhibits relatively high cohesion.

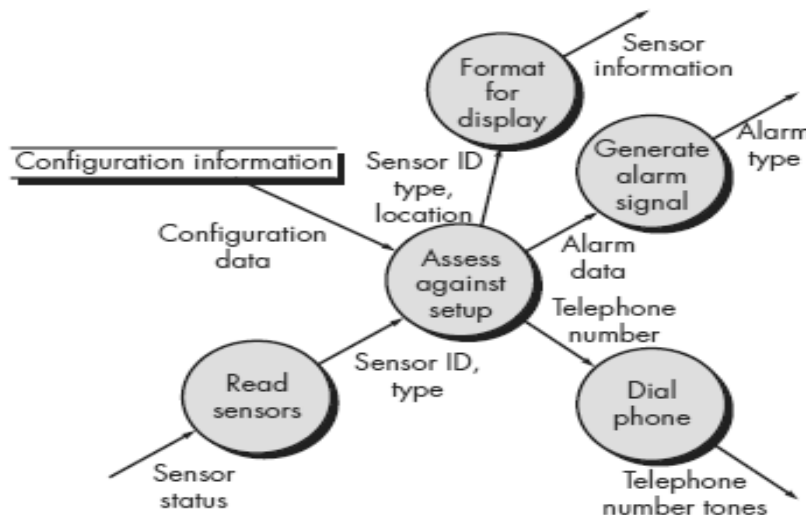


Fig :3 Level 2 DFD that refines the monitor sensors transform

Level 3 DFD for *monitor sensors* with flow boundaries

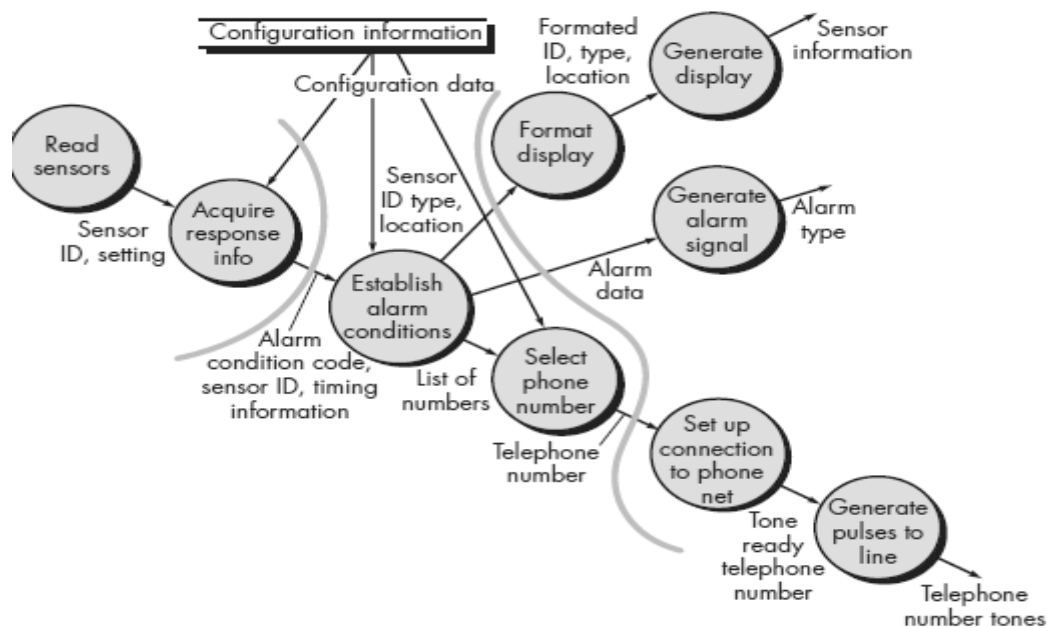
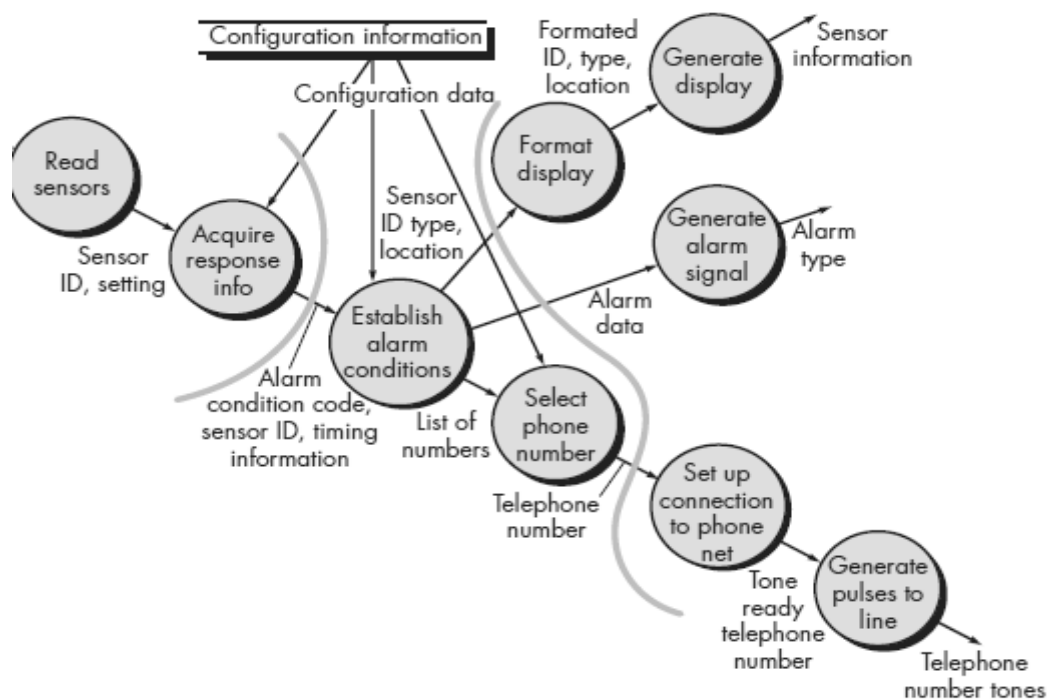


Fig:4 : Level 3

Level 3 DFD for *monitor sensors* with flow boundaries



Step 3. Determine whether the DFD has transform or transaction flow characteristics.

Evaluating the DFD (Figure 4), we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

**Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.** Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 4.

The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to read just a boundary (e.g., an incoming flow boundary separating read sensors and acquire response info could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

**Step 5. Perform “first-level factoring.”** The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and low level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work. When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing.

This first-level factoring for the monitor sensors subsystem is illustrated in Figure 5. A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:

- An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.

- A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

- An outgoing information processing controller, called alarm output controller, coordinates production of output information.

Although a three-pronged structure is implied by Figure 5, complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good functional independence characteristics.

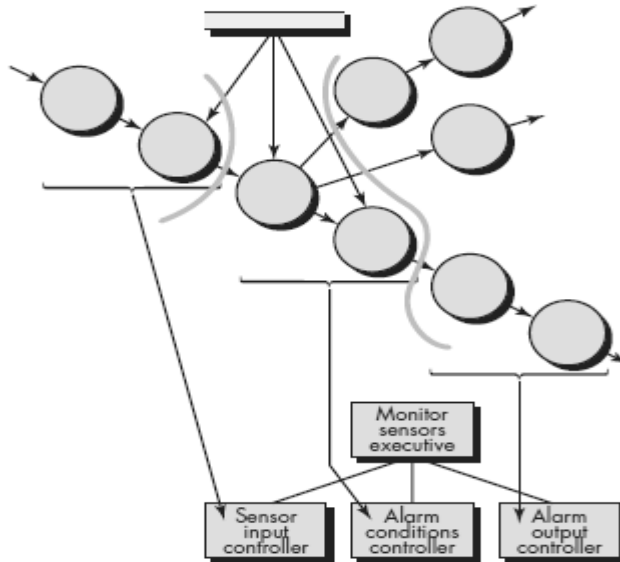
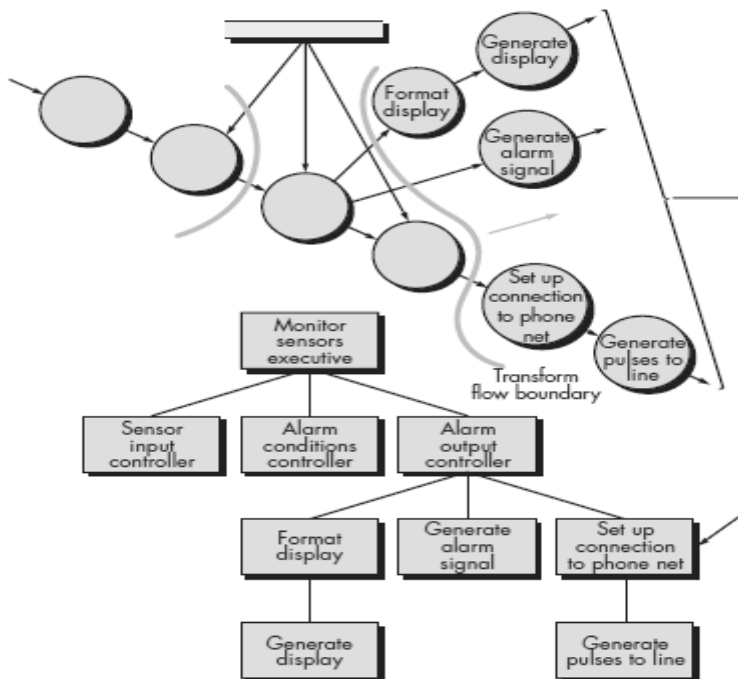


Fig : 5 First-level factoring for monitor sensors

**Step 6. Perform “second-level factoring.”** Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second level factoring is illustrated in Figure 6



Although Figure 6 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles

can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measures

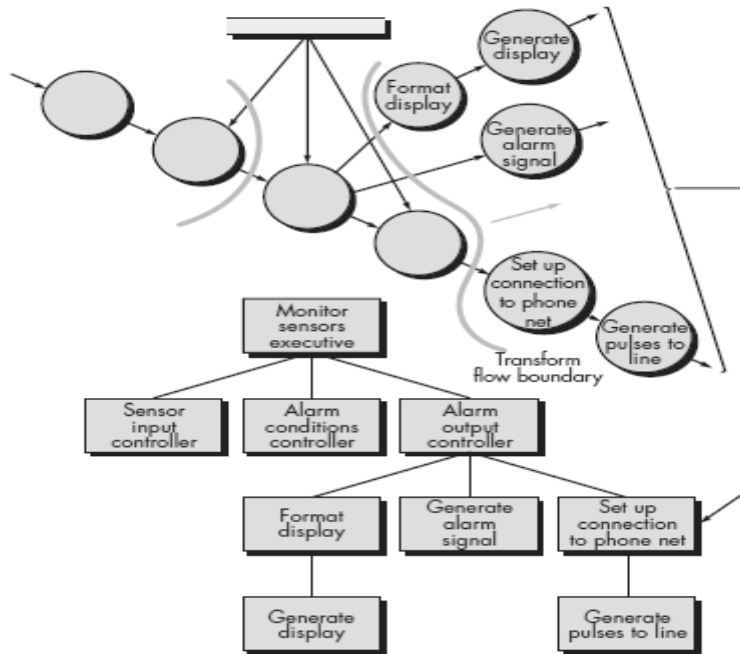


Fig : 6 Second-level factoring for monitor sensors

of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a “first-iteration” design.

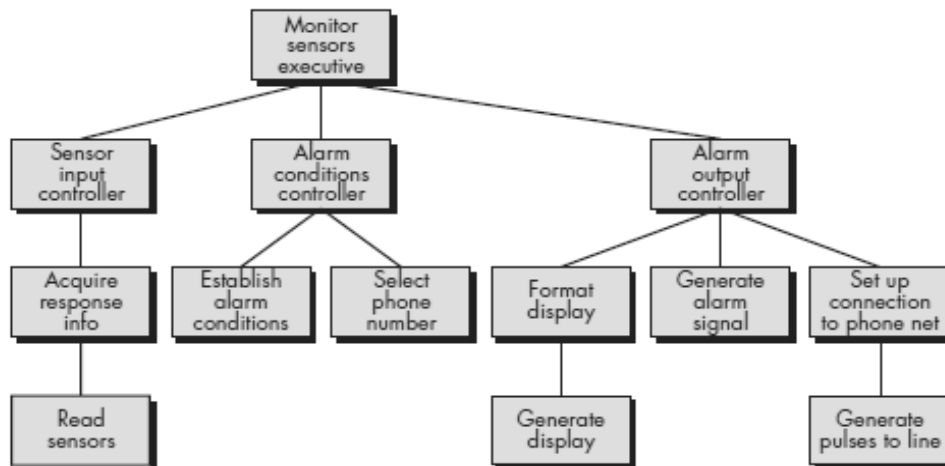
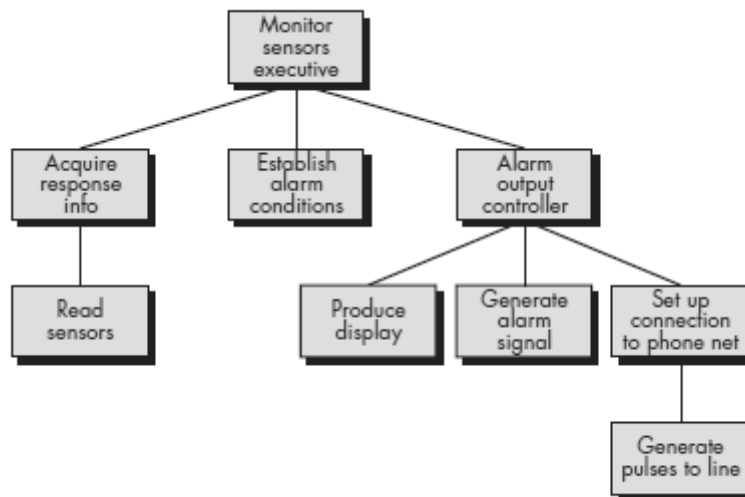


Fig : 7 First-iteration structure for monitor sensor

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of monitor sensors subsystem software is mapped somewhat differently. Each of the data conversion or

calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. Completed first-iteration architecture is shown in Figure 7. The components

**Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.** A first-iteration architecture can always be refined by applying concepts of functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a component that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter.



Refined program structure for monitor sensors

## COHESION AND COUPLING

**4. Describe the concepts of cohesion and coupling. State difference between cohesion and coupling with a suitable examples.** May: 03 , 08, 15,16.

### Cohesion

- Cohesion is the “single-mindedness’ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible.
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)  
Functional
- A module performs one and only one computation and then returns a result.

## Layer

- A higher layer component accesses the services of a lower layer component

## Communicational

- All operations that access the same data are defined within one class

## Kinds of cohesion

### **Sequential cohesion**

Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations

### **Procedural cohesion**

Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them

### **Temporal cohesion**

Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected

### **Utility cohesion**

Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

## **Coupling**

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
- **Data coupling** • Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling.

- **Stamp coupling**

A whole data structure or class instantiation is passed as a parameter to an operation

- **Control coupling** Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B(). Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result

- **Common coupling**

A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects

- **Content coupling**

One component secretly modifies data that is stored internally in another component

- **Subroutine call coupling**

When one operation is invoked it invokes another operation within side of it

• **Type use coupling**

Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration • If/when the type definition changes, every component that declares a variable of that data type must also change.

S.No	Coupling	Cohesion
1.	Coupling represents how the modules are connected with other modules or with other modules or with the outside world.	In cohesion the cohesive module performs only one thing.
2.	With coupling interface complexity is decided.	With cohesion data hiding can be done.
3.	The goal of coupling is to achieve lowest coupling.	The goal of cohesion is to achieve high cohesion.
4.	Various types of coupling are : Data coupling , control coupling common coupling , content coupling.	Various types of cohesion are: Coincidental cohesion, logical cohesion , Temporal cohesion, Procedural cohesion.

## USER INTERFACE DESIGN

5. Explain the user interface design activities.

May:07,16

Explain interface design activities. What steps do we perform to accomplish interface design?

May:07,Dec:08

List the activities of user interface design process

May:13

Explain golden rules of interface design.

May:09, Nov 16

User interface design creates an effective communication medium between a human and a computer.

**Three golden rules:**

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

**Place the User in Control**

Mandel defines a number of design principles that allow the user to maintain control:

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.

- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

### **Reduce the User's Memory Load**

Mandel defines design principles that enable an interface to reduce the user's memory load:

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive. The visual layout of the interface should be based on a real-world metaphor.

### **Make the Interface Consistent**

- Mandel defines a set of design principles that help make the interface consistent:
- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User interface design requires a good understanding of user needs. There are several phases and processes in the user interface design, some of which are more demanded upon than others, depending on the project.

□ Functionality requirements gathering – assembling a list of the functionality required by the system to accomplish the goals of the project and the potential needs of the users.

□ User and task analysis– a form of field research, it's the analysis of the potential users of the system by studying how they perform the tasks that the design must support, and conducting interviews to elucidate their goals.

Typical questions involve:

- o What would the user want the system to do?
- o How would the system fit in with the user's normal workflow or daily activities?
- o How technically savvy is the user and what similar systems does the user already use?
- o What interface look & feel styles appeal to the user?

□ Information architecture– development of the process and/or information flow of the system

□ Prototyping– development of wireframes, either in the form of paper prototypes or simple interactive screens. These prototypes are stripped of all look & feel elements and most content in order to concentrate on the interface.

□ Usability inspection– letting an evaluator inspect a user interface. This is generally considered to be cheaper to implement than usability testing (see step below), and can be used early on in the development process since it can be used to evaluate prototypes or specifications for the system, which usually can't be tested on users. Some common usability inspection methods include cognitive walkthrough, which

focuses the simplicity to accomplish tasks with the system for new users, heuristic evaluation, in which a set of heuristics are used to identify usability problems in the UI design, and pluralistic walkthrough, in which a selected group of people step through a task scenario and discuss usability issues.

□ Usability testing – testing of the prototypes on an actual user—often using a technique called think aloud protocol where you ask the user to talk about their thoughts during the experience. User interface design testing allows the designer to understand the reception of the design from the viewer's standpoint, and thus facilitates creating successful applications.

□ Graphical user interface design– actual look and feel design of the final graphical user interface (GUI). It may be based on the findings developed during the user research, and refined to fix any usability problems found through the results of testing.]

### **Interface analysis, Interface Design**

User interface design (UID) or user interface engineering is the design of websites, computers, appliances, machines, mobile communication devices, and software applications with the focus on the user's experience and interaction. The goal of user interface design is to make the user's interaction as simple and efficient as possible, in terms of accomplishing user goals—what is often called user-centered design.

Good user interface design facilitates finishing the task at hand without drawing unnecessary attention to itself. Graphic design and typography are utilized to support its usability, influencing how the user performs certain interactions and improving the aesthetic appeal of the design; design aesthetics may enhance or detract from the ability of users to use the functions of the interface. [1] The design process must balance technical functionality and visual elements (e.g., mental model) to create a system that is not only operational but also usable and adaptable to changing user needs.

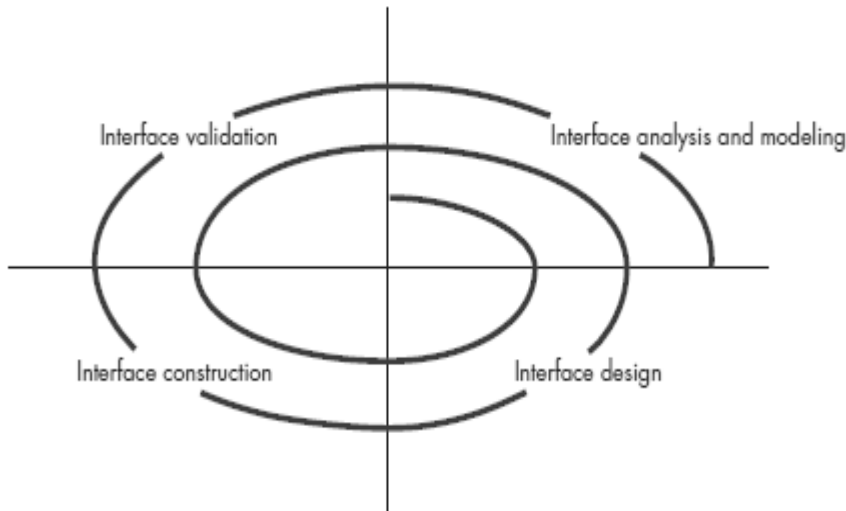
Interface design is involved in a wide range of projects from computer systems, to cars, to commercial planes; all of these projects involve much of the same basic human interactions yet also require some unique skills and knowledge. As a result, designers tend to specialize in certain types of projects and have skills centered on their expertise, whether that be software design, user research, web design, or industrial design.

Interface design deals with the process of developing a method for two (or more) modules in a system to connect and communicate. These modules can apply to hardware, software or the interface between a user and a machine. An example of a user interface could include a GUI, a control panel for a nuclear power plant,]or even the cockpit of an aircraft .In systems engineering, all the inputs and outputs of a system, subsystem, and its components are often listed in an interface control document as part of the requirements of the engineering project.

**User Interface design activities:**

User interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities:

- (1) Interface analysis and modeling,
- (2) interface design,
- (3) Interface construction, and
- (4) Interface validation.



**Interface analysis** focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system

**Interface design** is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

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

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

### **User Analysis**

Information from a broad array of sources can be used to accomplish this:

**User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

**Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

**Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

**Support input.** Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

## **Task Analysis and Modeling**

**Use cases.** Task analysis, the use case is developed to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first-person. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers.

**Workflow analysis.** When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows you to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs.

**User interface design models** have been proposed, all suggest some combination of the following steps:

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

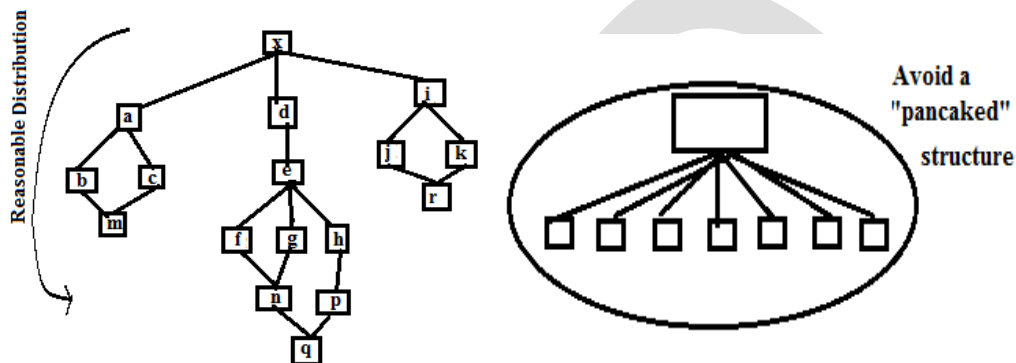
## DESIGN HEURISTICS

### 6. Discuss the design heuristics for effective modularity design.(8m) (4m)

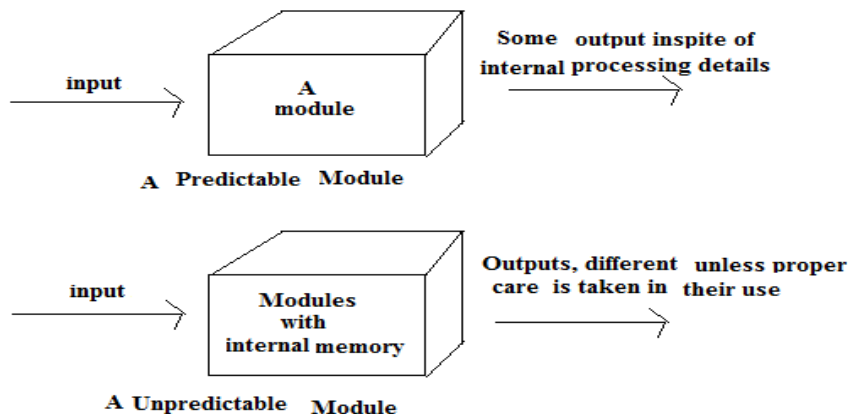
May/June 2013,16

#### Design Heuristics

1. Evaluate the “first iteration of the program Structure to reduce coupling and improve cohesion. The task is to improve module independence, once the program structure has been developed.
2. Attempt to minimize structures with fan-out. Strive for fan-in as depth increases.



3. Keep the scope of effect of a module within the scope of control of that module.
  - If module **e** makes a decision that affects module **r**, then the heuristic is violated, because module **r** lies outside the scope of control of module **e**.
4. Evaluate module interfaces to reduce complexity and redundancy to improve consistency.
5. Define modules whose function is predictable. But avoid modules that are overly restrictive.



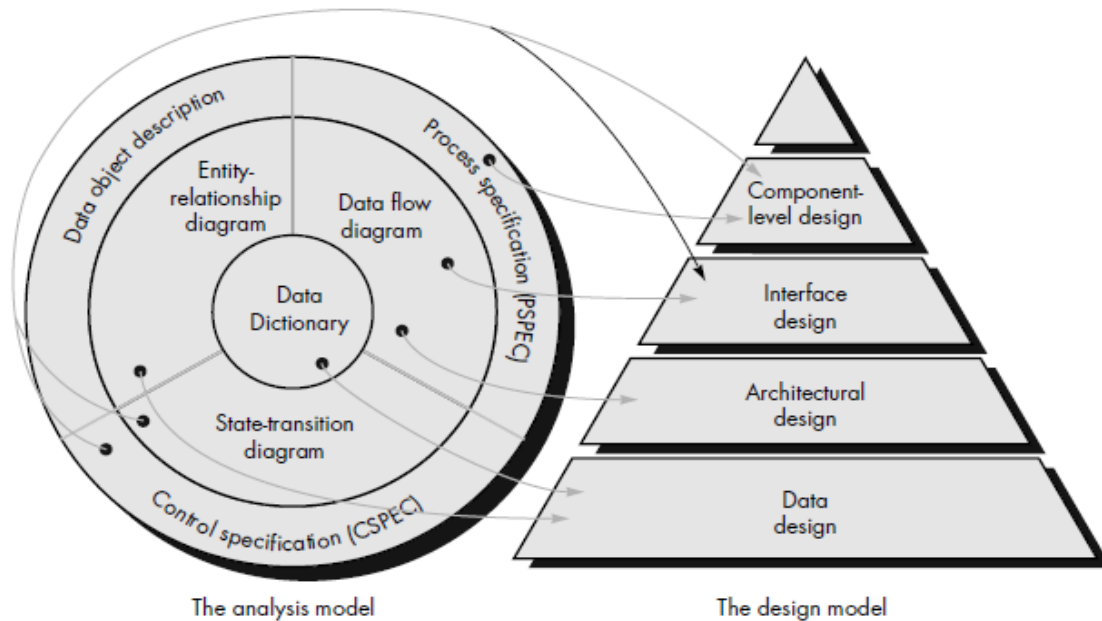
6. Strive for “controlled entry” modules by avoiding “pathological” connections”.

## PART- C

**1. What are the characteristics of good design? Describe the types of coupling and cohesion. How is design evaluation performed (APR/MAY/2010) (or) Which is a measure of interconnection among modules in a program structure? Explain (NOV/DEC/2011) – Answer – Coupling**

### **Purpose of Design**

- Design is where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system
- The design model provides detail about the software data structures, architecture, interfaces, and components
- The design model can be assessed for quality and be improved before code is generated and tests are conducted
  - Does the design contain errors, inconsistencies, or omissions?
  - Are there better design alternatives?
  - Can the design be implemented within the constraints, schedule, and cost that have been established?
- A designer must practice diversification and convergence
  - The designer selects from design components, component solutions, and knowledge available through catalogs, textbooks, and experience
  - The designer then chooses the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling
  - Convergence occurs as alternatives are considered and rejected until one particular configuration of components is chosen
- Software design is an iterative process through which requirements are translated into a blueprint for constructing the software
  - Design begins at a high level of abstraction that can be directly traced back to the data, functional, and behavioral requirements
  - As design iteration occurs, subsequent refinement leads to design representations at much lower levels of abstraction



**FIGURE 13.1** Translating the analysis model into a software design

**Goals of a Good Design** -three characteristics that 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
  - It must also accommodate all of the implicit requirements desired by the customer
- The design must be a readable and understandable guide for those who generate code, and for those who test and support the software
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective

**Technical criteria for good design**

- 1) A design should exhibit an architecture that
  - a) Has been created using recognizable architectural styles or patterns
  - b) Is composed of components that exhibit good design characteristics
  - c) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing
- 2) A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- 3) A design should contain distinct 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

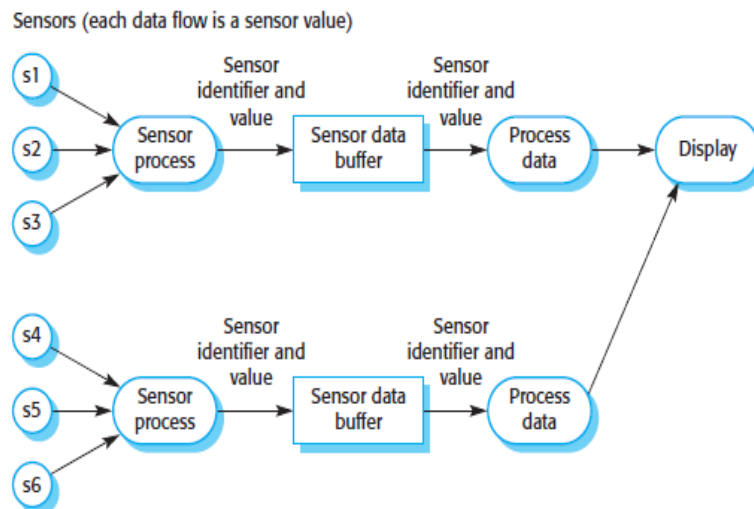
- 7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
- 8) A design should be represented using a notation that effectively communicates its meaning

**2. Write down software design procedures for data acquisition and control system (APR/MAY/2010) (NOV/DEC/2013). OR**

**Explain the various steps involved in analyzing and designing a data acquisition system**

- Data acquisition systems collect data from sensors for subsequent processing and analysis. These systems are used in circumstances where the sensors are collecting lots of data from the system's environment and it isn't possible or necessary to process the data collected in real-time. Data acquisition systems are commonly used in scientific experiments and process control systems where physical processes, such as a chemical reaction, happen very quickly.
- In data acquisition systems, the sensors may be generating data very quickly, and the key problem is to ensure that a sensor reading is collected before the sensor value changes. This leads to a generic architecture. The essential feature of the architecture of data acquisition systems is that each group of sensors has three processes associated with it.
- These are the sensor process that interfaces with the sensor and converts analogue data to digital values if necessary, a buffer process, and a process that consumes the data and carries out further processing. Sensors, of course, can be of different types, and the number of sensors in a group depends on the rate at which data arrives from the environment.
- In Figure below shows two groups of sensors, s1–s3 and s4–s6. I have also shown, on the right, a further process that displays the sensor data. Most data acquisition systems include display and reporting processes that aggregate the collected data and carry out further processing.

Figure 15.11 The generic architecture of data acquisition systems



- As an example of a data acquisition system, consider the system model shown in Figure below this represents a system that collects data from sensors monitoring the neutron flux in a nuclear reactor. The sensor data is placed in a buffer from which it is extracted and processed, and the average flux level is displayed on an operator's display.
- Each sensor has an associated process that converts the analogue input flux level into a digital signal. It passes this flux level, with the sensor identifier, to the sensor data buffer. The process responsible for data processing takes the data from this buffer, processes it and passes it to a display process for output on an operator console.
- In real-time systems that involve data acquisition and processing, the execution speeds and periods of the acquisition process (the producer) and the processing process (the consumer) may be out of step. When significant processing is required, the data acquisition may go faster than the data processing. If only simple computations need be carried out, the processing may be faster than the data acquisition.

### **3. Explain about component level design with example.**

**Nov : 16**

#### **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

### **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 than 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
- External coupling – occurs when a components communications or collaborates with infrastructure components (e.g. database)  
Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

#### **4.Discuss the differences between Function Oriented and Object Oriented Design. [May 2016]**

Differences between Function Oriented and Object Oriented Design:

1. FOD : The basic abstraction, which are given to the user, are real world functions.

OOD : The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.

2. FOD : Functions are grouped together by which higher level function is example of this technique is SA/SD.

OOD : Functions are grouped together on the basis of data they operate since the classes are associated with their methods.

3. FOD : In this approach the state information is often represented in a centralized shared memory.

OOD : In this approach the state information is not represented in a centralized memory but is implemented among the objects of the system.

4. FOD is mainly used for computation sensitive application.

OOD is mainly used for evolving system which mimicks a business process.

5. FOD : We decompose in function / procedure level.

OOD : We decompose in class level.

6. FOD : Begins by considering the use case diagrams and scenarios.

OOD : Begins by identifying objects and classes.

7. FOD : Top down approach.

OOD : Bottom up approach.

## UNIT- IV TESTING AND IMPLEMENTATION

Software testing fundamentals-Internal and external views of Testing-white box testing- basis path testing-control structure testing-black box testing- Regression Testing – Unit Testing – Integration Testing – Validation Testing – System Testing and Debugging – Software Implementation Techniques: Coding practices-Refactoring.

### PART – A

**1. Distinguish between verification and validation Dec: 07,16 May: 09, 13, 14**

Verification	Validation
Verification refers to the set of activities that ensure software correctly implements the specific function.	Validation refers to the set of activities that ensure that the software that has been built is traceable to customer requirements.
Are we building the product right?	Are we building the right product?
After a valid and complete specification the verification starts.	Validation begins as soon as project starts.
The verification is conducted to ensure whether software meets specification or not.	Validation is conducted to show that the user requirements are getting satisfied.

**2. Mention any two characteristics of software testing. OR**

**Write down generic characteristics of software testing. May: 08, 13**

1. Testing should be conducted by the developer, as well as by some independent group.
2. The testing must begin at the component level and must work outwards towards the integration testing.
3. Different testing must be appropriate times.

**3. What is the difference between alpha testing and beta testing? May: 09**

Alpha Testing	Beta Testing
This testing is done by a developer or by a customer under the supervision of developer in company's premises.	This testing is done by customer without any interference of developer and is done at customer's place.
Sometime full product is not tested using alpha testing and only core functionalities are tested.	The complete product is tested under this testing. Such product is usually given as free trial version.

#### **4. Write the best practices for coding.**

Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software.

- Clear definition of purpose.
- Simplicity of use.
- Reliability.
- Extensibility in the light of experience.
- Efficiency (fast enough for the purpose to which it is put).
- Minimum cost to develop.
- Clear, accurate, and precise user documents.

#### **5. Distinguish between stress and load testing.**

May: 12

Load testing is conducted to measure system performance based on volume of users. On the other hand the stress testing is conducted for measuring the breakpoint of a system when extreme load is applied. In load testing the expected load is applied to measure the performance of the system whereas in stress testing the extreme load is applied.

#### **6. What is a Big-Bang approach?**

Dec: 12

Big-Bang approach is used in non incremental integration testing. In this approach of integration testing, all the components are combined in advance and then entire program is tested as a whole.

#### **7. How is the software testing results related to the reliability of the software?**

Dec: 12

During the software testing the program is executed with the intention of finding as much errors as possible. The test cases are designed that are intended to discover yet-undiscovered errors. Thus after testing, majority of serious errors are removed from the system and it is turned into the model that satisfied user requirements.

#### **8. What are the levels at which the testing is done?**

Dec: 13

Testing can be done in the two levels such as:

- i)Component level testing: In the component level testing the individual component is tested.
- ii)System level testing: In system testing, the testing of group of components integrated to create a system or subsystem is done.

#### **9. What are the classes of loops that can be tested?**

May: 14

1. Simple loop
2. Nested Loop
3. Concatenated Loop
4. Unstructured Loop

#### **10. What is the need for regression testing ?**

May:15

Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes. Regression testing is a normal part of the program development process and, in larger companies, is done by code testing specialists.

**11. In unit testing of a module, it is found for a set of test data, at maximum 90% of the code alone were tested with the probability of success 0.9. What is the reliability of the module?**

The reliability of the module is at the most 0.81. For the given set of test data.

The 90% of the code is tested and reliability

of success is given as 0.9 i.e 90%

Hence, Reliability =  $(90/100) \times 0.9 = 0.81$

**12. How can refactoring be made more effective?**

**May : 16**

- Refactoring is the technique used to improve the code and avoid the design decay with time.
- The main risk if refactoring is that existing working code may break due to changes being made.

**13. Why does software fail it has passed from acceptance testing? May : 16**

Acceptance testing is usually done in cooperation with the customer, or by an internal customer proxy (product owner). For this type of testing we use test cases that cover the typical scenarios under which we expect the software to be used. This test must be conducted in a "production-like" environment, on hardware that is the same as, or close to, what a customer will use.

**14. What methods are used for breaking very long expression and statements?**

**Nov : 16**

Unit testing is used for breaking very long expression and statements. Because Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

## **PART – B**

### **WHITE BOX TESTING**

**1. Illustrate white box testing.**

**May: 04, 07, Dec: 07, May: 15**

White box testing:

- White-box testing of software is predicated on close examination of procedural detail.
- Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops.
- The "status of the program" may be examined at various points.
- White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases.
- Using this method, SE can derive test cases that

- Guarantee that all independent paths within a module have been exercised at least once
- Exercise all logical decisions on their true and false sides,
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity.
- Basis path testing:
- Basis path testing is a white-box testing technique
- To derive a logical complexity measure of a procedural design.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time.

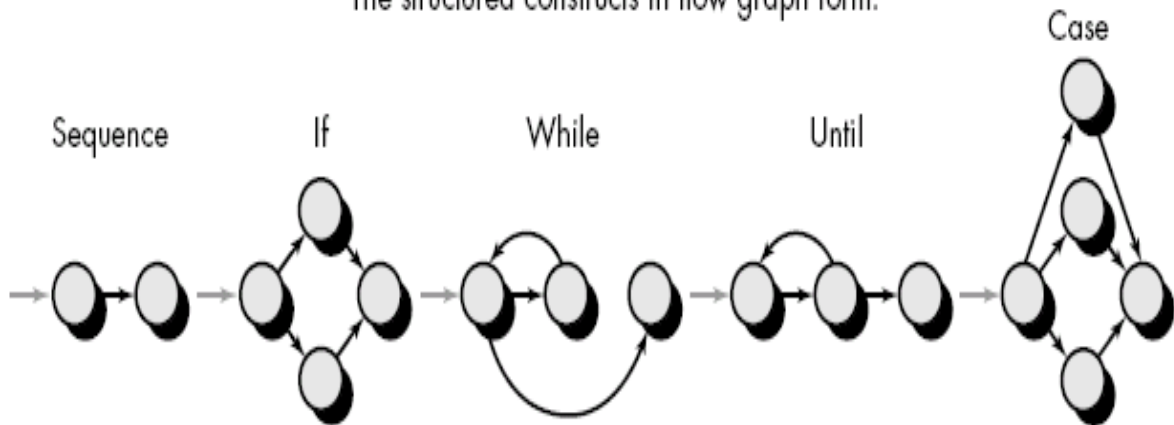
Methods:

1. Flow graph notation
2. Independent program paths or Cyclomatic complexity
3. Deriving test cases
4. Graph Matrices

Flow Graph Notation:

- Start with simple notation for the representation of *control flow* (called *flow graph*). It represent logical control flow.

The structured constructs in flow graph form:



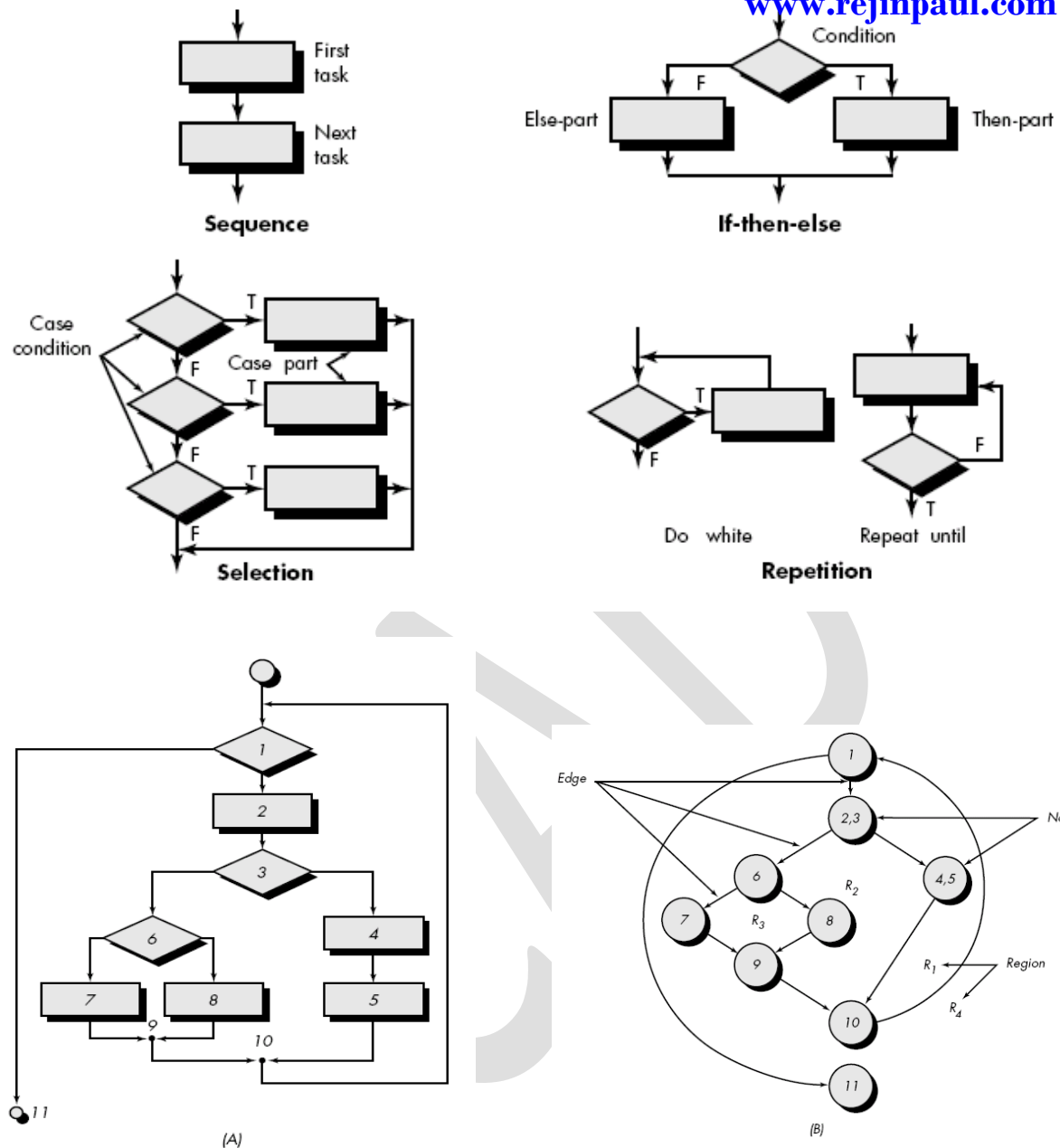
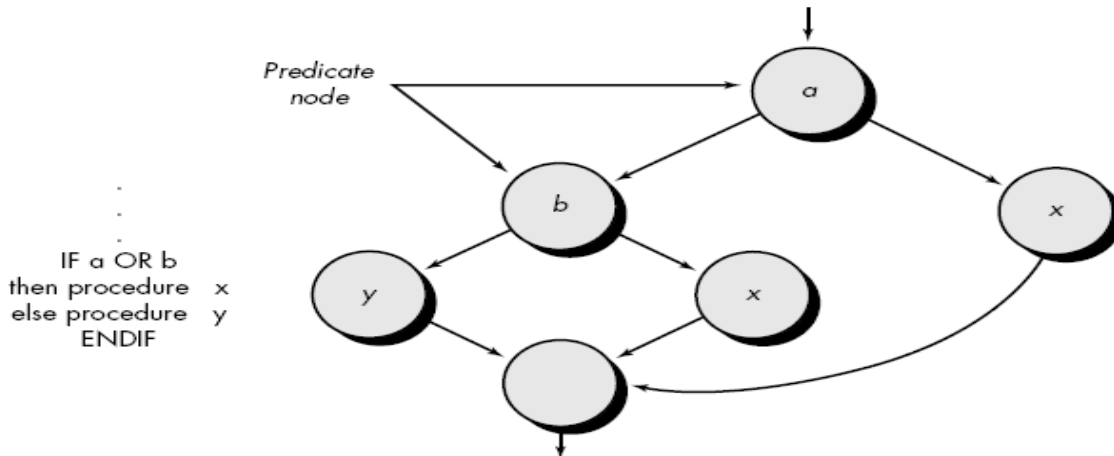


Fig. A represent program control structure and fig. B maps the flowchart into a corresponding flow graph.

In fig. B each circle, called flow graph node, represent one or more procedural statement.

- A sequence of process boxes and decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are parallel to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statement.

- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.
- When compound conditions are encountered in procedural design, flow graph becomes slightly more complicated.



- When we translating PDL segment into flow graph, separate node is created for each condition.
- Each node that contains a condition is called predicate node and is characterized by two or more edges comes from it.

Independent program paths or Cyclomatic complexity:

- An independent path is any path through the program that introduces at least one new set of processing statement or new condition.
- For example, a set of independent paths for flow graph:
  - ☐ Path 1: 1-11
  - ☐ Path 2: 1-2-3-4-5-10-1-11
  - ☐ Path 3: 1-2-3-6-8-9-1-11
  - ☐ Path 4: 1-2-3-6-7-9-1-11
- Note that each new path introduces a new edge.
- The path 1-2-3-4-5-10-1-2-3-6-8-9-1-11 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.
- Test cases should be designed to force execution of these paths (basis set).
- Every statement in the program should be executed at least once and every condition will have been executed on its true and false.
- Cyclomatic complexity is a software metrics that provides a quantitative measure of the logical complexity of a program.
- It defines no. of independent paths in the basis set and also provides number of test that must be conducted.
- One of three ways to compute cyclomatic complexity:

1. The no. of regions corresponds to the cyclomatic complexity.

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.

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 edges.

So the value of  $V(G)$  provides us with upper bound of test cases.

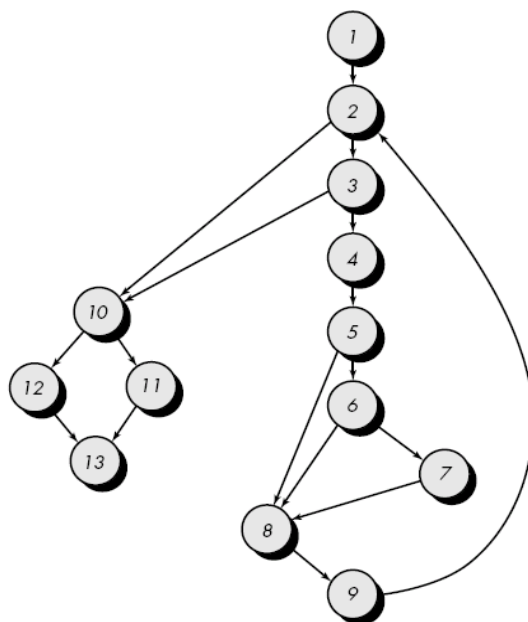
Deriving Test Cases:

- It is a series of steps method.
- The procedure average depicted in PDL.
- Average, an extremely simple algorithm, contains compound conditions and loops.

To derive basis set, follow the steps.

1. Using the design or code as a foundation, draw a corresponding flow graph.

A flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph node.



PROCEDURE average;

\* This procedure computes the average of 100 or fewer numbers that lie between bounding values; it also computes the sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;

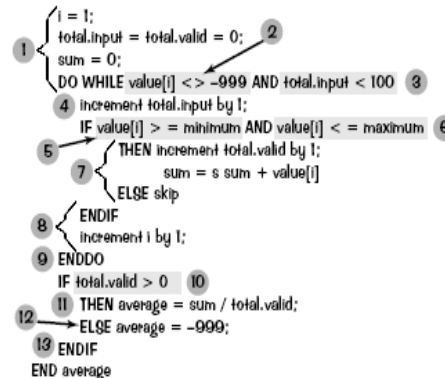
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;

TYPE average, total.input, total.valid;

minimum, maximum, sum IS SCALAR;

TYPE i IS INTEGER;



## BLACK BOX TESTING METHODS

### 2. Explain the various types of black box testing methods. Dec: 07,16 May: 15

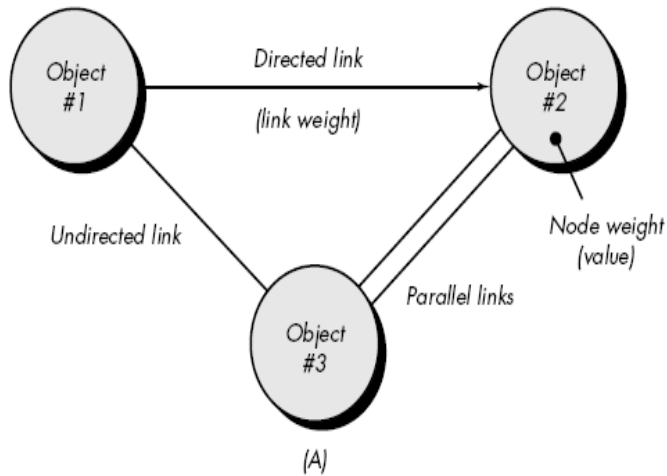
Black box testing:

- Also called behavioral testing, focuses on the functional requirements of the software.
- It 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 but it is complementary approach.

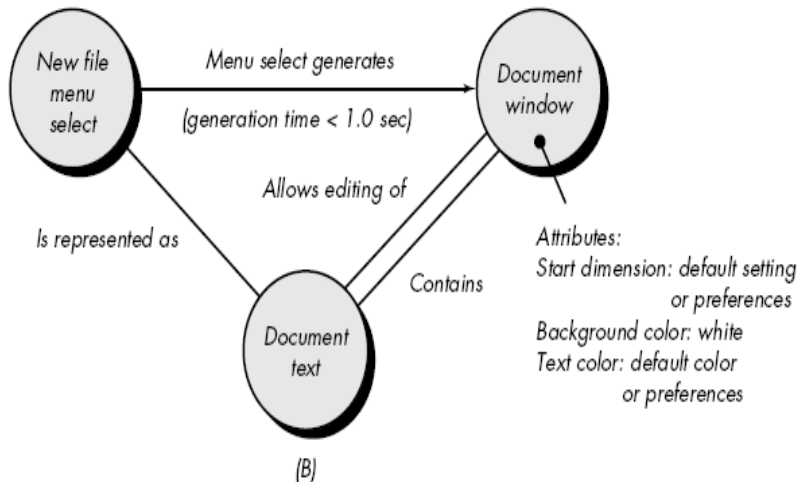
- Black-box testing attempts to find errors in the following categories:
  - ✓ Incorrect or missing functions,
  - ✓ Interface errors,
  - ✓ Errors in data structures or external data base access.
  - ✓ Behavior or performance errors,
  - ✓ Initialization and termination errors.
- Black-box testing purposely ignored 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?
- By applying black-box techniques, we derive a set of test cases that satisfy the following criteria
  - ✓ Test cases that reduce the number of additional test cases that must be designed to achieve reasonable testing (i.e minimize effort and time)
  - ✓ Test cases that tell us something about the presence or absence of classes of errors
- Black box testing methods
  - ✓ Graph-Based Testing Methods
  - ✓ Equivalence partitioning
  - ✓ Boundary value analysis (BVA)
  - ✓ Orthogonal Array Testing

#### Graph-Based Testing Methods:

- ✓ To understand the objects that are modeled in software and the relationships that connects these objects.
- ✓ Next step is to define a series of tests that verify “all objects have the expected relationship to one another.
- ✓ Stated in other way:
  - Create a graph of important objects and their relationships
  - Develop a series of tests that will cover the graph
- ✓ So that each object and relationship is exercised and errors are uncovered.
- ✓ Begin by creating graph –
  - ☐ a collection of nodes that represent objects
  - ☐ links that represent the relationships between objects
  - ☐ node weights that describe the properties of a node
  - ☐ link weights that describe some characteristic of a link.



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



- Object #1 = new file menu select
- Object #2 = document window
- Object #3 = document text
- Referring to example figure, a menu select on new file generates a document window.
- The link weight indicates that the window must be generated in less than 1.0 second.
- The node weight of document window provides a list of the window attributes that are to be expected when the window is generated.

- An undirected link establishes a symmetric relationship between the new file menu select and document text,
- parallel links indicate relationships between document window and document text
- Number of behavioral testing methods that can make use of graphs:
- Transaction flow modeling.
  - The nodes represent steps in some transaction and the links represent the logical connection between steps
- Finite state modeling.
  - The nodes represent different user observable states of the software and the links represent the transitions that occur to move from state to state. (Starting point and ending point)
- Data flow modeling.
  - The nodes are data objects and the links are the transformations that occur to translate one data object into another.
  - 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.

#### Equivalence Partitioning:

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
- An equivalence class represents a set of valid or invalid states for input conditions.
- Typically, an input condition is a specific numeric value, a range of values, a set of related values, or a Boolean condition.
- To define equivalence classes follow the guideline
- If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
- If an input condition requires a specific *value*, one valid and two invalid equivalence classes are defined.
- If an input condition specifies a member of a *set*, one valid and one invalid equivalence class are defined.
- If an input condition is Boolean, one valid and one invalid class are defined.
- Example:

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  
area code:

- Input condition, Boolean—the area code may or may not be present.
- Input condition, value— three digit number

prefix:

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

Suffix:

- 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— check, deposit, bill pay.

Boundary Value Analysis (BVA):

- 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.
- In other word, Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA

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. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary

Orthogonal Array Testing:

- 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 (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation .
- However, as the number of input values grows and the number of discrete values for each data item increases (exhaustive testing occurs)
- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- Orthogonal Array Testing can be used to reduce the number of combinations and provide maximum coverage with a minimum number of test cases.

Example:

- Consider the send function for a fax application.
- Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values.
- P1 takes on values:
  - P1 = 1, send it now
  - P1 = 2, send it one hour later
  - P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.
- OAT is an array of values in which each column represents a Parameter - value that can take a certain set of values called levels.
- Each row represents a test case.
- Parameters are combined pair-wise rather than representing all possible combinations of parameters and levels

Test case	Test parameters			
	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

## VARIOUS TESTING STRATEGY

**3. Explain about various testing strategy. May: 05, 06, Dec: 08, May: 10, 13**

Unit Testing Details:

- Interfaces tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis path testing should be used.
- All error handling paths should be tested.
- Drivers and/or stubs need to be developed to test incomplete software.

### 1. Unit Testing:

In unit testing the individual components are tested independently to ensure their quality. The focus is to uncover the errors in design and implementation.

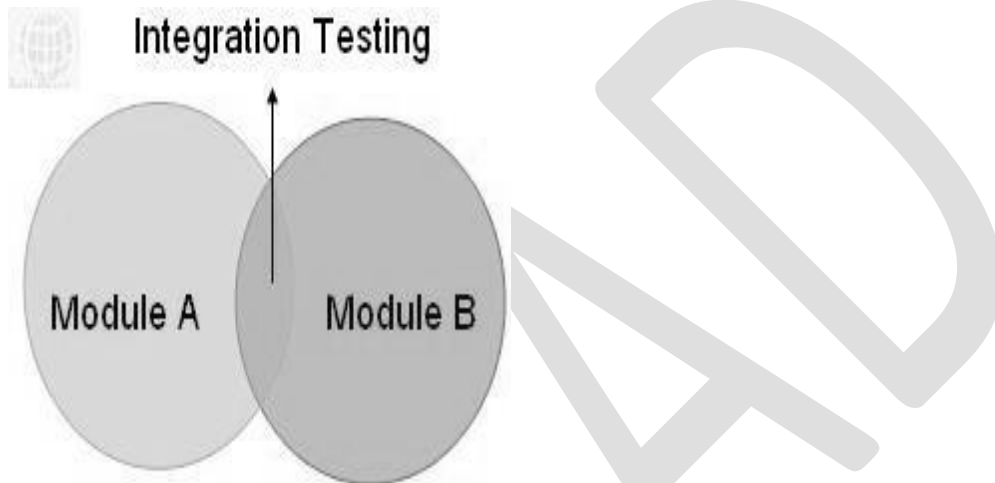
The various tests that are conducted during the unit test are described as below –

1. Module interfaces are tested for proper information flow in and out of the program.
2. Local data are examined to ensure that integrity is maintained.
3. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.

4. All the basis (independent) paths are tested for ensuring that all statements in the module have been executed only once.
5. All error handling paths should be tested.

## 2. Integration Testing:

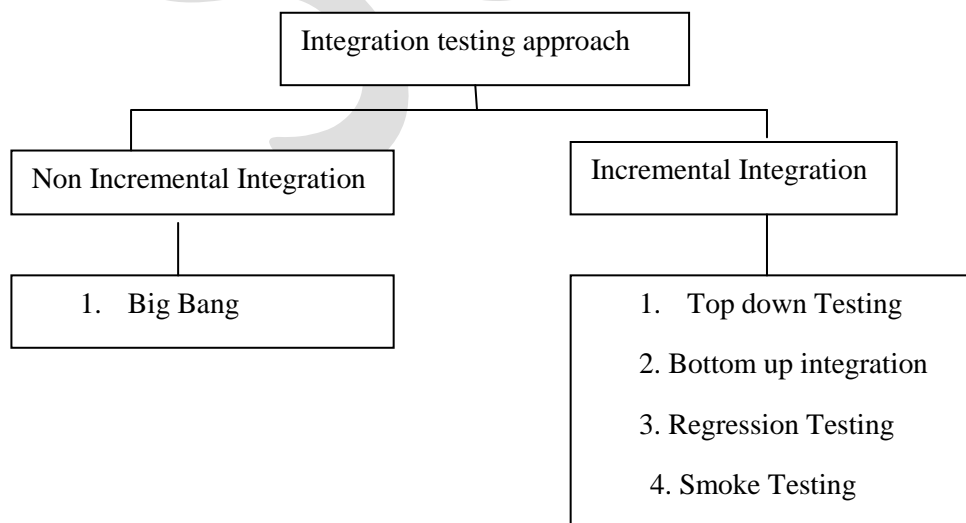
Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems. Also after integrating two different components together we do the integration testing. As displayed in the image below when two different modules 'Module A' and 'Module B' are integrated then the integration testing is done.



A group of dependent components are tested together to ensure their quality of their integration unit.

The objective is to take unit tested components and build a program structure that has been dictated by software design.

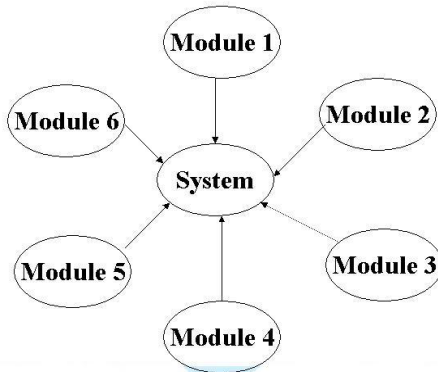
The integration testing can be carried out using two approaches.



### Big Bang integration testing:

In Big Bang integration testing all components or modules are integrated simultaneously, after which everything is tested as a whole. As per the below image all the modules from 'Module' to 'Module 6' are integrated simultaneously then the testing is carried out.

**Big Bang Integration Testing**



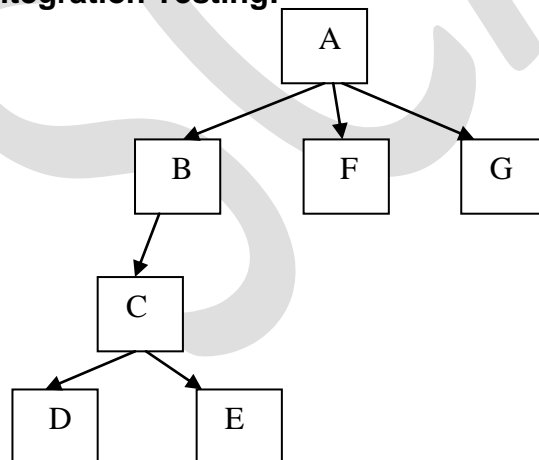
### Advantage:

Big Bang testing has the advantage that everything is finished before integration testing starts.

### Disadvantage:

The major disadvantage is that in general it is time consuming and difficult to trace the cause of failures because of this late integration.

### Top-Down Integration Testing:



- Is an incremental approach in which modules are integrated by moving down through the control structure.

Top town Integration process can be performed using following steps.

1. Main program used as a test driver and stubs are substitutes for components directly subordinate to it.

2. Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests and other stub is replaced with a real component.
5. Regression testing may be used to ensure that new errors not introduced.

#### **Advantages of Top-Down approach:**

The tested product is very consistent because the integration testing is basically performed in an environment that almost similar to that of reality

Stubs can be written with lesser time because when compared to the drivers then Stubs are simpler to author.

#### **Disadvantages of Top-Down approach:**

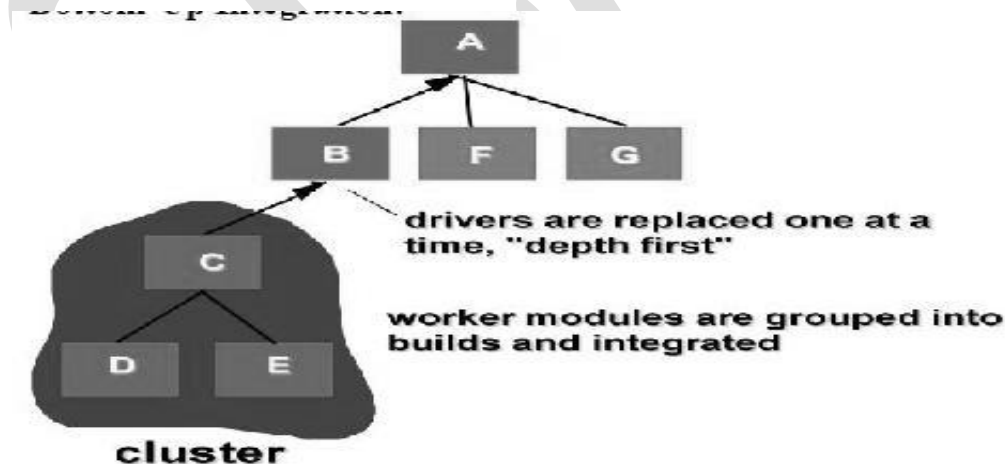
Basic functionality is tested at the end of cycle

#### **Bottom-Up Integration:**

In Bottom-Up Integration the modules at the lowest levels are integrated first, then integration is done by moving upward through the control structure.

Bottom up Integration process can be performed using following steps.

- Low level components are combined in clusters that perform a specific software function.
- A driver (control program) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.



#### **Advantage of Bottom-Up approach:**

- In this approach development and testing can be done together so that the product or application will be efficient and as per the customer specifications.

#### **Disadvantages of Bottom-Up approach:**

- We can catch the Key interface defects at the end of cycle
- It is required to create the test drivers for modules at all levels except the top control

### **3. Regression Testing:**

- The selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software. Also referred to as verification testing, regression testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.
- Regression test suit contains 3 different classes of test cases
  - ☐ Representative sample of existing test cases is used to exercise all software functions.
  - ☐ Additional test cases focusing software functions likely to be affected by the change.
  - ☐ Tests cases that focus on the changed software components.

### **4. Smoke Testing:**

Is a kind of integration testing technique used for time critical projects wherein the projects needs to be assessed on frequent basis.

Following activities need to be carried out in smoke testing:

- Software components already translated into code are integrated into a build.
- A series of tests designed to expose errors that will keep the build from performing its functions are created.
- The build is integrated with the other builds and the entire product is smoke tested daily using either top-down or bottom integration.

### **5. Validation Testing**

- ☐ Ensure that each function or performance characteristic conforms to its specification.
- ☐ Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, catalogued, and documented to allow its support during its maintenance phase.

### **6. Acceptance Testing:**

- Is a kind of testing conducted to ensure that the software works correctly for intended user in his or her normal work environment.

Types of acceptance testing are:

- Alpha test

Is a testing in which the version of the complete software is tested by customer under the supervision of the developer at the developer's site.

- Beta test

Is a testing in which the version of the complete software is tested by customer at his or her own site without the developer being present

## **7. System Testing:**

The system test is a series of tests conducted to fully the computer based system. Various types of system tests are:

- Recovery testing

Is intended to checks system's ability to recover from failures

- Security testing

Security testing verifies that system protection mechanism prevents improper penetration or data alteration

- Stress testing

Determines breakpoint of a system to establish maximum service level. Program is checked to see how well it deals with abnormal resource demands

## **8. Performance testing**

Performance testing evaluates the run-time performance of software.

Performance Testing:

- Stress test.
- Volume test.
- Configuration test (hardware & software).
- Compatibility.
- Regression tests.

Security tests.

- Timing tests.
- Environmental tests.
- Quality tests.
- Recovery tests.
- Maintenance tests.
- Documentation tests.
- Human factors tests.

Testing Life Cycle:

- Establish test objectives.
- Design criteria (review criteria).
- ☐ Correct.
- ☐ Feasible.
- ☐ Coverage.
- ☐ Demonstrate functionality.

## SOFTWARE TESTING PRINCIPLES

### 4. State the software testing principles

May: 08 ,Dec:09,16

**Why does software testing need extensive planning ? Explain. May : 2016**

Principles play an important role in all engineering disciplines and are usually introduced as part of an educational background in each branch of engineering.

#### **Principle: 1**

**Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.**

This principle supports testing as an execution-based activity to detect defects. It also supports the separation of testing from debugging since the intent of the latter is to locate defects and repair the software. The term “software component” is used in this context to represent any unit of software ranging in size and complexity from an individual procedure or method, to an entire software system. The term “defects” as used in this and in subsequent principles represents any deviations in the software that have a negative impact on its functionality, performance, reliability, security, and/or any other of its specified quality attributes

#### **Principle: 2**

**When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yet undetected defect(s).**

Principle 2 supports careful test design and provides a criterion with which to evaluate test case design and the effectiveness of the testing effort when the objective is to detect defects. It requires the tester to consider the goal for each test case, that is, which specific type of defect is to be detected by the test case. In this way the tester approaches testing in the same way a scientist approaches an experiment. In the case of the scientist there is a hypothesis involved that he/she wants to prove or disprove by means of the experiment.

#### **Principle: 3**

**Test results should be inspected meticulously.**

Testers need to carefully inspect and interpret test results. Several erroneous and costly scenarios may occur if care is not taken. For example: A failure may be overlooked, and the test may be granted a “pass” status when in reality the software has failed the test. Testing may continue based on erroneous test results. The defect may be revealed at some later stage of testing, but in that case it may be more costly and difficult to locate and repair.

- A failure may be suspected when in reality none exists. In this case the test may be granted a “fail” status. Much time and effort may be spent on trying to find the defect that does not exist. A careful reexamination of the test results could finally indicate that no failure has occurred. The outcome of a quality test may be misunderstood, resulting in unnecessary rework, or oversight of a critical problem.

**Principle: 4**

**A test case must contain the expected output or result.**

It is often obvious to the novice tester that test inputs must be part of a test case. However, the test case is of no value unless there is an explicit statement of the expected outputs or results, for example, a specific variable value must be observed or a certain panel button that must light up. Expected outputs allow the tester to determine (i) whether a defect has been revealed, and (ii) pass/fail status for the test. It is very important to have a correct statement of the output so that needless time is not spent due to is conceptions about the outcome of a test. The specification of test inputs and outputs should be part of test design activities.

**Principle: 5**

**Test cases should be developed for both valid and invalid input conditions.**

A tester must not assume that the software under test will always be provided with valid inputs. Inputs may be incorrect for several reasons For example; software users may have misunderstandings, or lack information about the nature of the inputs. They often make typographical errors even when complete/correct information is available. Devices may also provide invalid inputs due to erroneous conditions and malfunctions. Use of test cases that are based on invalid inputs is very useful for revealing defects since they may exercise the code in unexpected ways and identify unexpected software behavior. Invalid inputs also help developers and testers evaluate the robustness of the software, that is, its ability to recover when unexpected events occur.

**Principle: 6**

**The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.**

What this principle says is that the higher the number of defects already detected in a component, the more likely it is to have additional defects when it undergoes further testing. For example, if there are two components A and B, and testers have found 20 defects in A and 3 defects in B, then the probability of the existence of additional defects in A is higher than B. This empirical observation may be due to several causes. Defects often occur in clusters and often in code that has a high degree of complexity and is poorly designed.

**Principle: 7**

**Testing should be carried out by a group that is independent of the development group.**

This principle holds true for psychological as well as practical reasons. It is difficult for a developer to admit or conceive that software he/she has created and developed can be faulty. Testers must realize that (i) developers have a great deal of pride in their work, and (ii) on a practical level it may be difficult for them to conceptualize where defects could be found. Even when tests fail, developers often have difficulty in locating the defects since their mental model of the code may overshadow their view of code as it exists in actuality. They may also have

misconceptions or misunderstandings concerning the requirements and specifications relating to the software.

**Principle: 8**

**Tests must be repeatable and reusable.**

Principle 2 calls for a tester to view his/her work as similar to that of an experimental scientist. Principle 8 calls for experiments in the testing domain to require recording of the exact conditions of the test, any special events that occurred, equipment used, and a careful accounting of the results. This information is invaluable to the developers when the code is returned for debugging so that they can duplicate test conditions. It is also useful for tests that need to be repeated after defect repair.

**Principle: 9**

**Testing should be planned.**

Test plans should be developed for each level of testing, and objectives for each level should be described in the associated plan. The objectives should be stated as quantitatively as possible. Plans, with their precisely specified objectives, are necessary to ensure that adequate time and resources are allocated for testing tasks, and that testing can be monitored and managed.

**Principle: 10**

**Testing activities should be integrated into the software life cycle.**

It is no longer feasible to postpone testing activities until after the code has been written. Test planning activities as supported by Principle 10, should be integrated into the software life cycle starting as early as in the requirements analysis phase, and continue on throughout the software life cycle in parallel with development activities. In addition to test planning, some other types of testing activities such as usability testing can also be carried out early in the life cycle by using prototypes.

**Principle: 11**

**Testing is a creative and challenging task.**

Difficulties and challenges for the tester include the following:

- A tester needs to have comprehensive knowledge of the software engineering discipline.
- A tester needs to have knowledge from both experience and education as to how software is specified, designed, and developed.
- A tester needs to be able to manage many details.
- A tester needs to have knowledge of fault types and where faults of a certain type might occur in code constructs.
- A tester needs to reason like a scientist and propose hypotheses that relate to presence of specific types of defects.

## SAMPLE PROBLEM FOR CALCULATE THE CYCLOMATIC COMPLEXITY

5. Given a set of numbers 'n', the function FindPrime (a [],n) prints a number – if it is a prime number. Draw a control flow graph, calculate the cyclomatic complexity and enumerate all paths. State how many test case-s are needed to adequately cover the code in terms of branches, decisions and statement? Develop the necessary test cases using sample values for 'a' and 'n'.

Dec:13

The flow graph for prime numbers.

Number of edges  $E = 20$

Number of nodes  $N = 15$

Cyclomatic complexity  $= E - N + 2$

Path 1 : 1,2,3,4,5, ...15

Path 2: 1,2,15.

Path 3 : 1,2,3,4,5,6,7,10,...15.

Path 4 : 1,2,3,4,5,11,12,13,...15

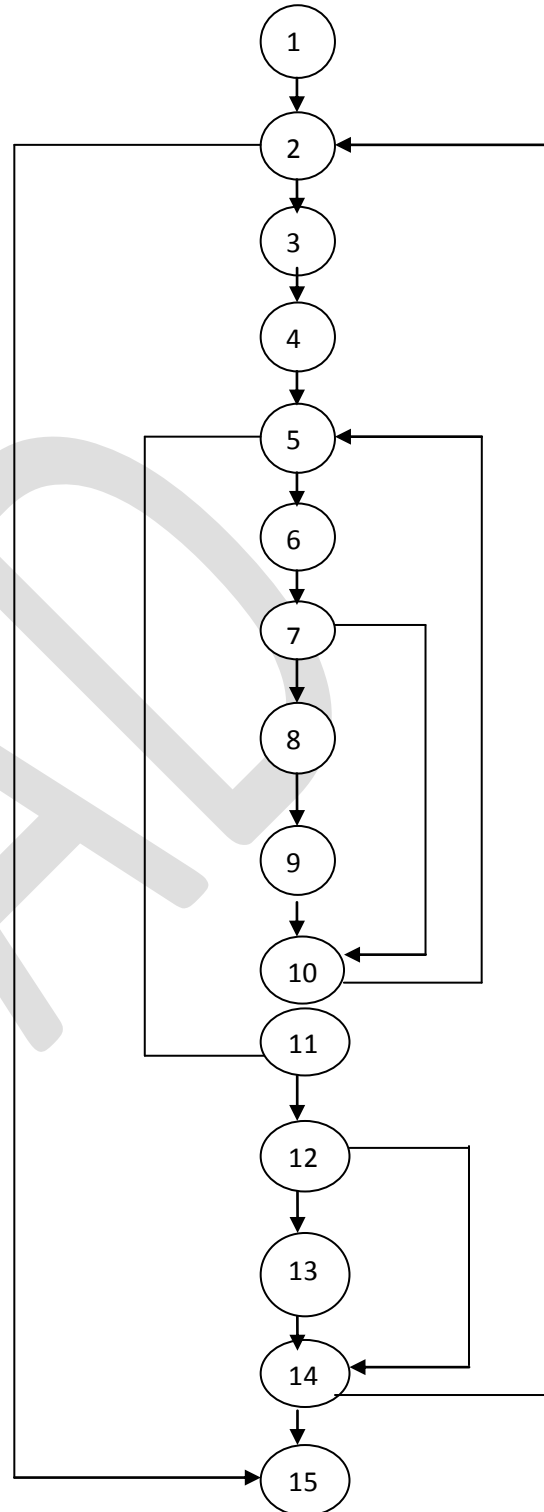
Path 5 : 1,2,3,...12,14,15.

Path 6 : 1,2,3,4,5,11,12,14,15.

Path 7 : 1,2,3,4,5,11,12,13,14,15.

```
void FindPrime(int a[], int n)
```

```
{  
    1. i=0;  
    2. while(i<n)  
        {  
    3. flag=0;  
    4. i=2;  
    5. while(j<a[i]) {  
    6. rem=a[i]%j;  
    7. if(rem==0)  
        {  
    8. flag=1;  
    9. break; }  
    10 j++;  
    11 } //end of while  
    12. if(flag==0)  
    13. Printf("%d",a[i]);  
    14. i++;  
    15. } //end of while  
}
```



Test Case:

Precondition :

1. a[] stores number to be tested.
2. j denotes any number within a range to test divisibility.

Test Case id	Test case name	Description	Step	Test case status(P/F)
1.	Divisibility by other number	Remainder is zero when number a[] is divisible by other than 1 or itself.	Rem = a[i]%j Where j<n If (rem==0) Set flag=1 If(flag!=0) then "Number is not Prime"	P
2.	Divisibility by 1 or self	Remainder is not zero when number a[] is not divisible by a number other than 1 or itself.	Rem = a[i]%j Where j<n If (rem!=0) Set flag=0 If(flag=0) then "Number is not Prime"	F

6.Consider the pseudocode for simple subtraction given below :[Nov / Dec 16]

(1) Program 'Simple Subtraction'

(2) Input (x,y)

(3) Output (x)

(4) Output (y)

(5) If x > y then DO

(6) x-y =z

(7) Else y-x = z

(8) EndIf

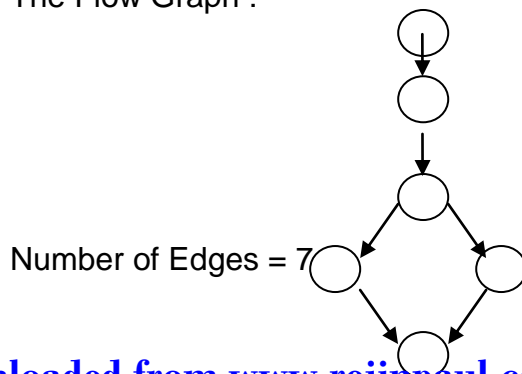
(9) Output (z)

(10) Output " End Program"

Perform basis path testing and generate test cases.

Solution :

The Flow Graph :



Number of Nodes = 7

Cyclomatic Complexity =  $E - N + 2$

$$= 7 - 7 + 2 = 2$$

Cyclomatic Complexity =  $P + 1$

$$= 1 + 1 = 2$$

**Basis Set of path**

Path 1 : 1,2,3,4,5,7

Path 2 : 1,2,3,4,6,7

**Test Cases :**

Test case id	Test case Name	Description	Step	Test case status (P/F)
1.	Checking list element with key	Check If x is greater than y or not .	If $x > y$	P
2.	Validating the output value	If condition is true then subtract x and y else subtract y and x.	If $x > y$ Do $x - y = z$ Else $y - x = z$	F

## VALIDATION TESTING

### 6. Explain validation testing in detail (8)

Requirements are validated against the constructed software

- Validation testing follows integration testing
  - The distinction between conventional and object-oriented software disappears
  - Focuses on user-visible actions and user-recognizable output from the system
  - Demonstrates conformity with requirements
  - Designed to ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
  - After each validation test
  - The function or performance characteristic conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created
- A configuration review or audit ensures that all elements of the software

configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

- **Alpha testing**
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- **Beta testing**
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base.

## **DEBUGGING PROCESS & CODING PROCESS**

7. i) Explain the debugging process in detail (8)

ii) Explain coding in detail (8)

### **Debugging**

- Debugging occurs as a consequence of successful testing
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction.

### **Coding Process**

It is the phase where the designed software project is implemented as coded program. **(ie), source code** which is compiled into the final working **software Application**.

### **Guidelines in brief**

A general overview of all of the above:

1. Know what the code block must perform
2. Indicate a brief description of what a variable is for (reference to commenting)
3. Correct errors as they occur.
4. Keep your code simple
5. Maintain naming conventions which are uniform throughout.

## **Coding standards**

"Establish programming conventions before you begin programming. It's nearly impossible to change code to match them later."

As listed near the end of Coding conventions, there are different conventions for different programming languages, so it may be counterproductive to apply the same conventions across different languages.

The use of coding conventions is particularly important when a project involves more than one programmer (there have been projects with thousands of programmers). It is much easier for a programmer to read code written by someone else if all code follows the same conventions.

For some examples of bad coding conventions, Roedy Green provides a lengthy (tongue-in-cheek) article on how to produce unmaintainable code.

## **Commenting**

Due to time restrictions or enthusiastic programmers who want immediate results for their code, commenting of code often takes a back seat. Programmers working as a team have found it better to leave comments behind since coding usually follows cycles, or more than one person may work on a particular module. However, some commenting can decrease the cost of knowledge transfer between developers working on the same module.

In the early days of computing, one commenting practice was to leave a brief description of the following:

1. Name of the module.
2. Purpose of the Module.
3. Description of the Module (In brief).
4. Original Author
5. Modifications
6. Authors who modified code with a description on why it was modified.

However, the last two items have largely been obsolete by the advent of revision control systems. Also regarding complicated logic being used, it is a good practice to leave a comment "block" so that another programmer can understand what exactly is happening.

Unit testing can be another way to show how code is intended to be used. Modifications and authorship can be reliably tracked using a source-code revision control system, rather than using comments.

## **Naming conventions**

Use of proper naming conventions is considered good practice. Sometimes programmers tend to use X1, Y1, etc. as variables and forget to replace them with meaningful ones, causing confusion.

In order to prevent this waste of time, it is usually considered good practice to use descriptive names in the code since we deal with real data.

Example: A variable for taking in weight as a parameter for a truck can be named TrkWeight or TruckWeightKilograms, with TruckWeightKilograms being the more preferable one, since it is instantly recognizable.

### **Keep the code simple**

The code that a programmer writes should be simple. Complicated logic for achieving a simple thing should be kept to a minimum since the code might be modified by another programmer in the future. The logic one programmer implemented may not make perfect sense to another. So, always keep the code as simple as possible.

### **Portability**

Program code should **never ever** contain "hard-coded", *i.e.* literal, values referring to environmental parameters, such as absolute file paths, file names, user names, host names, IP addresses, URLs, UDP/TCP ports. Otherwise the application will not run on a host that has a different design than anticipated. Such variables should be parametrized, and configured for the hosting environment outside of the application proper (e.g. property files, application server, or even a database).

### **Code Development**

#### **Code building**

A best practice for building code involves daily builds and testing, or better still continuous integration, or even continuous delivery.

#### **Testing**

Testing is an integral part of software development that needs to be planned. It is also important that testing is done proactively; meaning that test cases are planned before coding starts and test cases are developed while the application is being designed and coded.

#### **Debugging the code and correcting errors**

Programmers tend to write the complete code and then begin debugging and checking for errors. Though this approach can save time in smaller projects, bigger and complex ones tend to have too many variables and functions that need attention. Therefore, it is good to debug every module once you are done and not the entire program. This saves time in the long run so that one does not end up wasting a lot of time on figuring out what is wrong. Unit tests for individual modules, and/or functional tests for web services and web applications, can help with this.

#### **Deployment**

Deployment is the final stage of releasing an application for users.

## **REFACTORING**

### **8. Explain Refactoring in detail.**

May 2016, Nov 2016

#### **Refactoring is:**

- Restructuring (rearranging) code...

- ...in a series of small, semantics-preserving transformations (i.e. the code keeps working).....in order to make the code easier to maintain and modify
- we need to keep the code working
- we need small steps that preserve semantics
- we need to have unit tests to prove the code works

**You should refactor:**

- Any time that you see a better way to do things.
- “Better” means making the code easier to understand and to modify in the future
- You can do so without breaking the code

**Unit tests are essential for this**

- switch statements are very rare in properly designed object-oriented code
- Therefore, a switch statement is a simple and easily detected “bad smell”
- Of course, not all uses of switch are bad
- A switch statement should not be used to distinguish between various kinds of object

There are several well-defined refactoring for this case

The simplest is the creation of subclasses

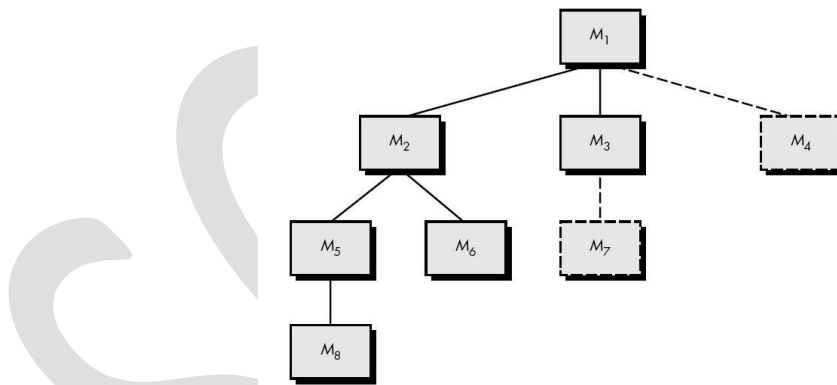
Basic	Improved
<pre>class Animal {     final int MAMMAL = 0, BIRD = 1,            REPTILE = 2;     int myKind; // set in constructor     ... String getSkin()     {         switch (myKind)         {             case MAMMAL: return "hair";             case BIRD: return "feathers";             case REPTILE: return "scales";             default: return "integument";         }     } }</pre>	<pre>class Animal {     String getSkin() { return "integument"; } class Mammal extends Animal {     String getSkin() { return "hair"; } } class Bird extends Animal {     String getSkin() { return "feathers"; } } class Reptile extends Animal {     String getSkin() { return "scales"; } }</pre>

## PART - C

### 1. Distinguish between top-down and bottom-up integration. (10)(AUC DEC 2010)

#### Top-down integration

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to Figure 18.6, depth-first integration would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.



From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

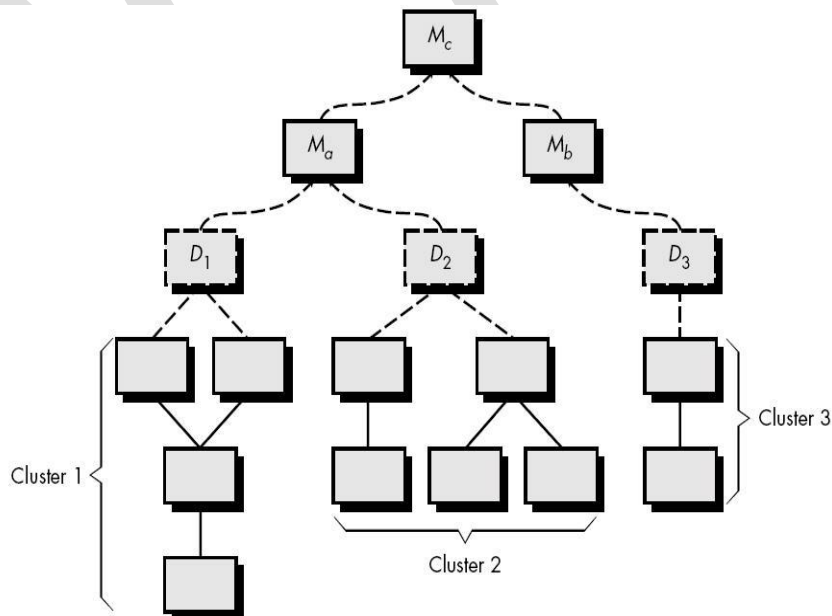
## Bottom-up Integration

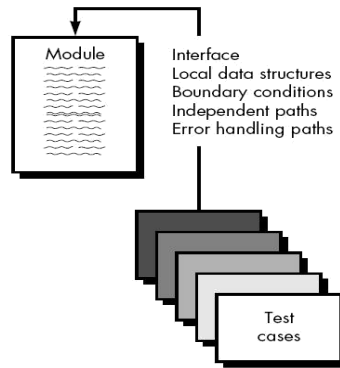
- Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).
- Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to  $M_a$ . Drivers  $D_1$  and  $D_2$  are removed and the clusters are interfaced directly to  $M_a$ . Similarly, driver  $D_3$  for cluster 3 is removed prior to integration with module  $M_b$ . Both  $M_a$  and  $M_b$  will ultimately be integrated with component  $M_c$ , and so forth.





S.No	Top-Down Integration	Bottom-up integration
1	A top-down approach is essentially the breaking down of a system to gain insight into its compositional sub-systems.	A bottom-up approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system.
2	An overview of the system is formulated, specifying but not dealing any first-level subsystems.	The individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems.
3	It is often specified with the assistance of “black boxes”, these make it easier to manipulate.	It is often resembles a “seed model”, whereby the beginnings are small but eventually grow in complexity and completeness.
4	It emphasize planning and a complete understanding of the system.	Zero redundancy
5	Easy to visualize functionality	Good unit test case can be written to validate changes.
6	Sense of completeness in the requirement	Developer has only option to use unit testing tools to test the logic.
7	Easy to show the progress of development	Easy to manage changes and modification.
8	High possibility of redundancy	Effort involved writing cases.

## 2. What is unit testing? Why is it important? Explain the unit test considerations and test procedures. (10) (AUC MAY 2011)

### UNIT TESTING

- Unit testing focuses verification effort on the smallest unit of software design—the software component or module.
- Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing.
- The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

#### • Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure. The module interface is tested to ensure that information properly flows into and out of the program unit under test.

The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.

Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.

All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

Basis path and loop testing are effective techniques for uncovering a broad array of path errors. Among the more common errors in computation are

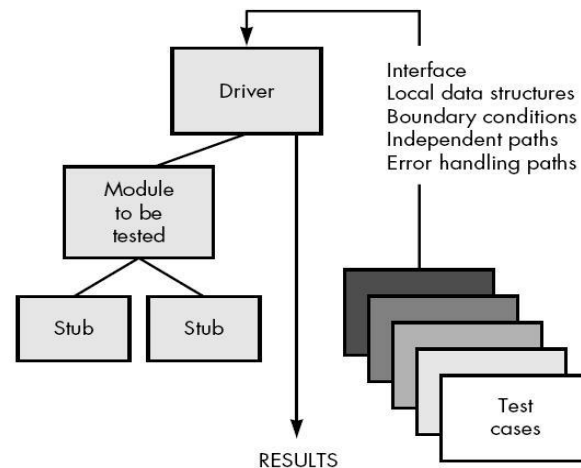
- (1) misunderstood or incorrect arithmetic precedence.
- (2) mixed mode operations.
- (3) incorrect initialization.
- (4) precision inaccuracy.
- (5) Incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another. Test cases should uncover errors such as

- (1) comparison of different data types,
- (2) incorrect logical operators or precedence,
- (3) expectation of equality when precision error makes equality unlikely,
- (4) incorrect comparison of variables,
- (5) improper or nonexistent loop termination,
- (6) failure to exit when divergent iteration is encountered, and
- (7) Improperly modified loop variables.

Among the potential errors that should be tested when error handling is evaluated are

1. Error description is unintelligible.
2. Error noted does not correspond to error encountered.
3. Error condition causes system intervention prior to error handling.
4. Exception-condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.



## Unit Test Procedures

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins.

A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure .In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (called by) the component to be tested.

A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software.

In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

### 3 . Explain in detail about Integration testing. (May/Jun 2014)

A group of dependent components are tested together to ensure their quality of their integration unit.

- The objective is to take unit tested components and build a program structure that has been dictated by software design.

- The focus of integration testing is to uncover errors in
- Design and construction of software architecture.
- Integrated functions or operations at subsystem level.
- Interfaces and interactions between them.
- Resource integration and/or environment integration.

The integration testing can be carried out using two approaches.

1. The non-incremental integration
2. Incremental integration

Non -incremental integration    Incremental integration    Top down testing

- Big bang

Bottom up integration    Regression testing    Smoke testing

- The non-incremental integration is given by the “big bang” approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results. A set of errors is tested as a whole. Correction is difficult because isolation of causes is complicated by the size of the entire program. Once these errors are corrected new ones appear. This process continues infinitely.

Advantage of big-bang:

This approach is simple.

Disadvantages:

- It is hard to debug.
- It is not easy to isolate errors while testing.

In this approach it is not easy to validate test results.

- An incremental construction strategy includes

- 1) Top down integration
- 2) Bottom up integration
- 3) Regression testing
- 4) Smoke testing

**4.Consider a program for determining the previous date. Its input is a triple of day ,month and year with the values in the range  $1 \leq \text{month} \leq 12$  ,  $1 \leq \text{day} \leq 31$  ,  $1990 \leq \text{year} \leq 2014$ . The possible outputs would be previous date or invalid input date. Design the boundary value test cases. [ 8 Marks ][May / june 2016]**

Problem domain : “ Next Date “ is a function consisting of three variables like :month , date and year.It returns the date of next day as output. It reads current date as input date. The conditions are

C1 :  $1 \leq \text{month} \leq 12$

C2 :  $1 \leq \text{day} \leq 31$

C3 :  $1999 \leq \text{year} \leq 2014$

If any one condition out of C1, C2 or fails, then this function produces an output “ value of month not in range 1...12”.

Many combinations of dates can exist, hence we can simply display one message for this function : “Invalid Input Date”.

Complexities in Next Date Functions :

A Year is called as a leap year if it is divisible by 4 , unless it is a century year.Century years are leap years only if they are multiples of400.So, 1992,196 and 2000 are leap years while 1900 is not a leap year.

The next Date program takes date as input and checks it for validity. If it is valid, it returns the next date as its output.

Here , the value of  $n = 3$ .

BVA yields  $(4n+1)$  test cases according to single fault assumption theory. So , the total number of test cases will be  $(4*3+1) = 12 + 1 = 13$ .

Test case ID	Month (mm)	Date (dd)	Year (yyyy)	Expected output
1	6	15	1900	6
2	6	15	1901	6
3	6	15	1962	6
4	6	15	2024	6
5	6	15	2025	6
6	6	1	1962	6
7	6	2	1962	6
8	6	30	1962	6
9	6	31	1962	Invalid Data as june has 30 days
10	1	15	1962	1
11	2	15	1962	2
12	11	15	1962	11
13	12	15	1962	12

Where  $n = 3$

BVA yields  $= (4n + 1) = 4(3) + 1 = 13$  test cases.

## 5.Compare and contrast alpha and beta Testing. 2016

Alpha Testing	Beta Testing (Field Testing)
1. It is always performed by the developers at the software development site.	1. It is always performed by the customers at their own site.
2. Sometimes it is also performed by Independent Testing Team.	2. It is not performed by Independent Testing Team.
3. Alpha Testing is not open to the market and public	3. Beta Testing is always open to the market and public.
4. It is conducted for the software application and project.	4. It is usually conducted for software product.
5. It is always performed in Virtual Environment.	5. It is performed in Real Time Environment.
6. It is always performed within the organization.	6. It is always performed outside the organization.
7. It is the form of Acceptance Testing.	7. It is also the form of Acceptance Testing.
8. Alpha Testing is definitely performed and carried out at the developing organizations location with the involvement of developers.	8. Beta Testing (field testing) is performed and carried out by users or you can say people at their own locations and site using customer data.
9. It comes under the category of both White Box Testing and Black Box Testing.	9. It is only a kind of Black Box Testing.
10. Alpha Testing is always performed at the time of Acceptance Testing when developers test the product and project to check whether it meets the user requirements or not.	10. Beta Testing is always performed at the time when software product and project are marketed.
11. It is always performed at the developer's premises in the absence of the users.	11. It is always performed at the user's premises in the absence of the development team.
12. Alpha Testing is not known by any other different name.	12 Beta Testing is also known by the name Field Testing means it is also known as field testing.
13. It is considered as the User Acceptance Testing (UAT) which is done at developer's area.	13. It is also considered as the User Acceptance Testing (UAT) which is done at customers or users area.

## UNIT – V PROJECT MANAGEMENT

Estimation – FP Based, LOC Based, Make/Buy Decision, COCOMO II - Planning – Project Plan, Planning Process, RFP Risk Management – Identification, Projection, RMMM - Scheduling and Tracking –Relationship between people and effort, Task Set & Network, Scheduling, EVA – Process and Project Metrics.

### PART – A

#### 1. What are software planning activities?

May: 09

1. Identify the constraints in the project.
2. Make initial assessment of the project.
3. Define project milestones and deadlines.
4. While developing the project i) Prepare project schedule. ii) Review progress of the project.

#### 2. What are risk management activities?

May : 09

1. Risk Identification
2. Risk Analysis
3. Risk Planning
4. Risk Monitoring

#### 3. What information does a software project plan provide?

May: 10

1. The quality plan describes the quality procedures and standards that will be used in a project.
2. The validation plan describes the approach, resources and schedule required for system validation.
3. The configuration management procedures and structures used are also described by the project plan.
4. The maintenance requirements of the system, maintenance cost and effort requirement information is described by the software project plan.

#### 4. How is productivity and cost related to function points?

Nov : 16

- Using Function Points to Estimate Test Cases
- Using Function Points to help Understand Wide Productivity Ranges
- Using Function Points to help Understand Scope Creep
- Using Function Points to help Calculate the True Cost of Software
- Using Function Points to help estimate project cost, schedule and effort

#### 5. What is project planning? Highlight the activities in project planning

May: 12,15

Project planning is an activity in which the project plan is prepared. This plan is mainly concerned with schedule and budget.

- Initiation
- Planning
- Execution and Control
- Closure

**6. What do you mean by estimation risk?**

**May: 13**

Estimation risk is measured by the degree of uncertainty in the quantitative estimates for cost, schedule, and resources.

**7. What is error tracking?**

**May: 14**

Error tracking is a process of finding out and correcting the errors that may occur during the software development process at various stages such as software design, coding or documenting.

**8. How is the software risks assessed?**

**May:13**

Risk assessments assigns a risk rating or score to each information asset. While this number does not mean anything in absolute terms, it is useful in gauging the relative risk to each vulnerable information asset. Factors of risk are likelihood, value, current controls, and uncertainty.

**9. Define software measure.**

**May: 08**

Software measure is a numeric value for a attribute of a software product or process. There are two types of software measures – direct measure and indirect measure. The direct measure refers to immediately measureable attributes. For example, lines of code. The indirect measure refers to the aspects that are not immediately quantifiable or measurable. For example functionality of the program.

**10. Define software quality**

**Dec: 13**

The software quality can be defined as the degree to which a system component or process meets specific requirements.

**11. State the importance of scheduling activities in project planning.**

**April / May 2015**

In order to build a complex system, many software engineering tasks occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without a schedule. It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

**12. List a few process and product metrics.**

**May : 16**

**Process metrics** used to provide indicators that lead to long term process improvement. Project metrics enable project manager to

1. Assess status of ongoing project
2. Track potential risks
3. Uncover problem are before they go critical
4. Adjust work flow or tasks

5. Evaluate the project team's ability to control quality of software work products

**Product metrics:**

1. Aid in the evaluation of analysis and design models
2. Provide an indication of the complexity of procedural designs and source code
3. Facilitate the design of more effective testing techniques
4. Assess the stability of a fielded software product

**13. What is RMMM?**

The risk mitigation, monitoring, and management plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan. Once the RMMM has been documented and the project has begun risk mitigation and monitoring steps commence.

**14. Will exhaustive testing guarantee that the program is 100% correct?**

**May : 16**

No, because there are many times a program runs correctly as designed, but I think there is no such thing as 100% reliability even after very exhaustive testing. Many things within a person's computer can cause a program to not function as designed even if it works for most other users of that program.

**15. What is risk management?**

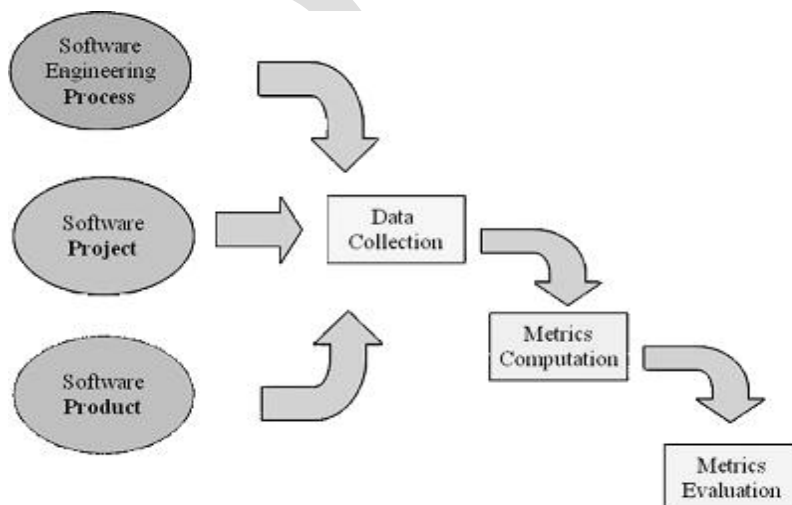
**Nov :16**

Risk management is an activity in which risks in the software projects are identified.

**PART – B  
PROCESS & PROJECT METRICS**

1. Describe two metrics which have been used to measure the software.

**May 04,05**



Software process and project metrics are quantitative measures. The software measures are collected by software engineers and software metrics are analyzed by software managers.

- They are a management tool.
- They offer insight into the effectiveness of the software process and the projects that are conducted using the process as a framework.
- Basic quality and productivity data are collected.
- These data are analyzed, compared against past averages, and assessed.
- The goal is to determine whether quality and productivity improvements have occurred.
- The data can also be used to pinpoint problem areas.
- Remedies can then be developed and the software process can be improved.

Use of Measurement:

- Can be applied to the software process with the intent of improving it on a continuous basis.
- Can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control.
- Can be used to help assess the quality of software work products and to assist in tactical decision making as a project proceeds.

**Reason for measure:**

- To characterize in order to
- Gain an understanding of processes, products, resources, and environments
- Establish baselines for comparisons with future assessments
- To evaluate in order to determine status with respect to plans
- To predict in order to gain understanding of relationships among processes and products
- Build models of these relationships
- To improve in order to Identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance

**Metric in Process Domain:**

- Process metrics are collected across all projects and over long periods of time.
- They are used for making strategic decisions.
- The intent is to provide a set of process indicators that lead to long-term software process improvement.
- The only way to know how/where to improve any process is to
- Measure specific attributes of the process
- Develop a set of meaningful metrics based on these attributes
- Use the metrics to provide indicators that will lead to a strategy for improvement

**Properties of Process Metrics**

- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics

- Don't use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
- Never use metrics to threaten individuals or teams
- Metrics data that indicate a problem should not be considered "negative"
  - Such data are merely an indicator for process improvement
- Don't obsess on a single metric to the exclusion of other important metrics

### **Metrics in Project Domain**

- Project metrics enable a software project manager to
  - Assess the status of an ongoing project
  - Track potential risks
  - Uncover problem areas before their status becomes critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software work products
- Many of the same metrics are used in both the process and project domain
- Project metrics are used for making tactical decisions
  - They are used to adapt project workflow and technical activities

### **Use of Project Metrics:**

- The first application of project metrics occurs during estimation
  - Metrics from past projects are used as a basis for estimating time and effort
- As a project proceeds, the amount of time and effort expended are compared to original estimates
- As technical work commences, other project metrics become important
  - Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)
  - Error uncovered during each generic framework activity (i.e, communication, planning, modeling, construction, deployment) are measured

### **Categories of Software Measurement**

- Two categories of software measurement
  - Direct measures of the
    - Software process (cost, effort, etc.)
    - Software product (lines of code produced, execution speed, defects reported over time, etc.)
  - Indirect measures of the
    - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)
- Project metrics can be consolidated to create process metrics for an organization

### **Size oriented metrics**

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value
- Metrics include

- o Errors per KLOC - Errors per person-month
- o Defects per KLOC - KLOC per person-month
- o Dollars per KLOC - Dollars per page of documentation
- o Pages of documentation per KLOC

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

Opponents argue that KLOC measurements

- o Are dependent on the programming language
- o Penalize well-designed but short programs
- o Cannot easily accommodate nonprocedural languages
- o Require a level of detail that may be difficult to achieve

### **Function oriented metrics**

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- Most widely used metric of this type is the function point:  $FP = \text{count total} * [0.65 + 0.01 * \text{sum (value adj. factors)}]$
- Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)
- Like the KLOC measure, function point use also has proponents and opponents
- Proponents claim that
  - o FP is programming language independent
  - o FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
- Opponents claim that
  - o FP requires some “sleight of hand” because the computation is based on subjective data
  - o Counts of the information domain can be difficult to collect after the fact
  - o FP has no direct physical meaning...it's just a number

### **Object Oriented Metrics:**

- Average number of support classes per key class
  - o Key classes are identified early in a project (e.g., at requirements analysis)
  - o Estimation of the number of support classes can be made from the number of key classes
  - o GUI applications have between two and three times more support classes as key classes
  - o Non-GUI applications have between one and two times more support classes as key classes

Number of subsystems

- o A subsystem is an aggregation of classes that support a function that is visible to the end user of a system

### **Metrics for Software Quality**

Correctness

o This is the number of defects per KLOC, where a defect is a verified lack of conformance to requirements

o Defects

Defects are those problems reported by a program user after the program is released for general use

o Maintainability

This describes the ease with which a program can be corrected if an error is found, adapted if the environment changes, or enhanced if the customer has changed requirements

o Mean time to change (MTTC) :

The time to analyze, design, implement, test, and distribute a change to all users  
Maintainable programs on average have a lower MTTC

### **Defect Removal Efficiency (DRE)**

DRE represents the effectiveness of quality assurance activities. The DRE also helps the project manager to assess the progress of software project as it gets developed through its scheduled work task. Any errors that remain uncovered and are found in later tasks are called defects.

The defect removal efficiency can be defined as

$DRE = E / (E + D)$  Where DRE is the defect removal efficiency, E is the error, D is the defect.

### **Measuring Quality**

Following are the measure of the software quality:

1. **Correctness:** Is a degree to which the software produces the desired functionality. The correctness can be measured as  $\text{Correctness} = \text{Defects per KLOC}$ .
2. **Integrity:** Integrity is basically an ability of the system to withstand against the attacks. There are two attributes that are associated with integrity: threat and security.
3. **Usability:** User friendliness of the system or ability of the system that indicates the usefulness of the system.
4. **Maintainability:** Is an ability of the system to accommodate the corrections made after encountering errors, adapting the environment changes in the system in order to satisfy the user.

## **CATEGORIES OF SOFTWARE RISKS**

**2. What are the categories of software risks? Give an overview about risk management.**

**May: 14**

Risk is a potential problem – it might happen and it might not conceptual definition of risk

o Risk concerns future happenings

o Risk involves change in mind, opinion, actions, places, etc.

o Risk involves choice and the uncertainty that choice entails

Two characteristics of risk

o Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)

o Loss – the risk becomes a reality and unwanted consequences or losses occur

### **Risk Categorization**

#### 1) Project risks

They threaten the project plan. If they become real, it is likely that the project schedule will slip and that costs will increase

#### 2) Technical risks

They threaten the quality and timeliness of the software to be produced. If they become real, implementation may become difficult or impossible

#### 3) Business risks

They threaten the viability of the software to be built. If they become real, they jeopardize the project or the product

Sub-categories of Business risks

i) Market risk – building an excellent product or system that no one really wants

ii) Strategic risk – building a product that no longer fits into the overall business strategy for the company

iii) Sales risk – building a product that the sales force doesn't understand how to sell

iv) Management risk – losing the support of senior management due to a change in focus or a change in people

v) Budget risk – losing budgetary or personnel commitment

#### 4. Known risks

Those risks 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)

#### 5. Predictable risks

Those risks that are extrapolated from past project experience (e.g., past turnover)

#### 6. Unpredictable risks

Those risks that can and do occur, but are extremely difficult to identify in advance

### **Risk Identification**

Risk identification is a systematic attempt to specify threats to the project plan. By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary

Generic risks

Risks that are a potential threat to every software project.

Product-specific risks

Risks that can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built. This requires examination of the project plan and the statement of scope. "What special characteristics of this product may threaten our project plan?"

## Risk Item Checklist

Used as one way to identify risks

Focuses on known and predictable risks in specific subcategories can be organized in several ways

- o A list of characteristics relevant to each risk subcategory
- o Questionnaire that leads to an estimate on the impact of each risk
- o A list containing a set of risk component and drivers and their probability of occurrence

## Known and Predictable Risk Categories

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

The project manager identifies the risk drivers that affect the following risk components

- o **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
- o **Cost risk** - the degree of uncertainty that the project budget will be maintained
- o **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
- o **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time

## Risk projection

Risk projection (or estimation) attempts to rate each risk in two ways

The probability that the risk is real. The consequence of the problems associated with the risk, should it occur. The project planner, managers, and technical staff perform four risk projection steps. The intent of these steps is to consider risks in a manner that leads to prioritization. By prioritizing risks, the software team can allocate limited resources where they will have the most impact

## Steps

1. Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2. Delineate the consequences of the risk

### 3. Estimate the impact of the risk on the project and product

#### **Risk Table**

A risk table provides a project manager with a simple technique for risk projection

It consists of five columns

Risk Summary – short description of the risk

Risk Category – one of seven risk categories

Probability – estimation of risk occurrence based on group input

Impact – (1) catastrophic (2) critical (3) marginal (4) negligible

RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

Risk Summary	Risk Category	Probability	Impact (1-4)	RMMM

#### **RISK MITIGATION, MONITORING, AND MANAGEMENT**

An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning. If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for risk mitigation. For example, assume that high staff turnover is noted as a project risk r1.

Based on past history and management intuition, the likelihood l1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule. To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:

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

As the project proceeds, risk-monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with

compensation and benefits, and the availability of jobs within the company and outside it are all monitored.

In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

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

In addition, you can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to speed.” Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.” This might include video-based knowledge capture, the development of “commentary documents or Wikis,” and/or meeting with other team members who will remain on the project.

It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost. For example, spending the time to back up every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, you perform a classic cost-benefit analysis.

If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is “backup,” management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

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

There are three issues in strategy for handling the risk is

- 1) Risk avoidance 2) Risk Monitoring 3) Risk management

### **Risk mitigation**

Risk mitigation means preventing the risks to occur. Following are the steps to be taken for mitigating the risks:

1. Communicate with the concerned staff to find of probable risk.
2. Find out and eliminate all those causes that can create risk before the project starts.
3. Conduct timely reviews in order to speed up the work.

### Risk Monitoring

The risk monitoring process following things must be monitored by the project manager,

1. The approach or the behavior of the team members as pressure of project varies.
2. The degree in which the team performs with the spirit of “team work”.
3. The type of co-operation among the team members.
4. The types of problems that are occurring.

### Risk Management

Project manager performs this task when risk becomes a reality. If project manager is successful in applying the project mitigation effectively then it becomes very much easy to manage the risks.

Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet. 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.

The format of the RIS is illustrated in Figure .Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As I have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking

Risk information sheet			
Risk ID: P02-432	Date: 5/9/09	Prob: 80%	Impact: high
<b>Description:</b> Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.			
<b>Refinement/context:</b> Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards. Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components. Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.			
<b>Mitigation/monitoring:</b> 1. Contact third party to determine conformance with design standards. 2. Press for interface standards completion; consider component structure when deciding on interface protocol. 3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.			
<b>Management/contingency plan/trigger:</b> RE computed to be \$20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly. Trigger: Mitigation steps unproductive as of 7/1/09.			
<b>Current status:</b> 5/12/09: Mitigation steps initiated.			
Originator: D. Gagne		Assigned: B. Laster	

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.

## FUNCTION POINT ANALYSIS

### 3. Describe function point analysis with a neat example. Dec:06 Nov: 10

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- Most widely used metric of this type is the function point:  $FP = \text{count total} * [0.65 + 0.01 * \text{sum (value adj. factors)}]$
- Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)
- Like the KLOC measure, function point use also has proponents and opponents
- Proponents claim that
  - o FP is programming language independent
  - o FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
  - o FP requires some “sleight of hand” because the computation is based on subjective data
  - o Counts of the information domain can be difficult to collect after the fact
  - o FP has no direct physical meaning...it's just a number

Function points are derived using

1. Countable measures of the software requirements domain.
2. Assessments of the software complexity.

#### Calculate Function Point

The data for following information domain characteristics are collected.

1. Number of user inputs – Each user input which provides distinct application data to the software is counted.
2. Number of user outputs – Each user output that provides application data to the user is counted, e.g –Screens, reports, error messages.
3. Number of user inquiries – An on-line input that results in the generation of some immediate software response in the form of an output.
4. Number of files – Each logical master file, i.e a logical grouping of data that may be part of a database or a separate file.
5. Number of external interfaces – All machine readable interfaces that are used to transmit information to another system are counted.

The organization needs to develop criteria which determine whether a particular entry is simple, average, or complex.

The weighting factors should be determined by observations or by experiments.

Domain Characteristics	Count		Weighting Factor			Count
			Simple	Average	Complex	
Number of user input		X	3	4	6	
Number of user output		X	4	3	7	
Number of user inquiries		X	3	4	6	
Number of fields		X	7	10	15	
Number of external interfaces		X	5	7	10	
Count totals						

The count table can be computed with the help of above table.

Now the software complexity can be computed by answering following questions.

These are complexity adjustment values.

- Rate the above factors according to the following scale:
- $\text{Function Points (FP)} = \text{Count total} \times (0.65 + (0.01 \times \text{Sum}(F_i)))$

Once the functional point is calculated then we can compute various measures as follows

- $\text{Productivity} = \text{FP} / \text{person} - \text{month}$
- $\text{Quality} = \text{Number of faults} / \text{FP}$
- $\text{Cost} = \$ / \text{FP}$
- $\text{Documentation} = \text{Pages of documentation} / \text{FP}$

Advantages:

1. This method is independent of programming languages.
2. It is based on the data which can be obtained in early stage of project.

Disadvantages:

1. Many aspects of this method are not validated.
2. The functional point has no significant meaning. It is numerical value.

3b ) Consider the following function point components and their complexity. If the total degree of influence is 52, find the estimated function points.

[Nov / Dec 2016]

Function Type	Estimated Count	Complexity
ELF	2	7
ILF	4	10
EQ	22	4
EO	16	5
EI	24	4

**Solution :**

$$FP = UFC \times VAF$$

Where ,      FP = Function Point

UFC = FP Count Total

VAF = Value Adjustment Factor

$$UFC = 2 \times 7 + 4 \times 10 + 22 \times 4 + 16 \times 5 + 24 \times 4 = 318$$

$$VAF = [0.65 + (0.01 \times \sum(F_i))]$$

$$= [0.65 + (0.01 \times 52)]$$

$$= 1.17$$

$$\text{So, FP Estimated} = (318 \times 1.17)$$

$$= 372$$

## COCOMO II MODEL

### 4. Explain in detail the COCOMO II Model

May: 08, Dec: 13, May: 14,16

**Constructive COst Model II (COCOMO® II)** is a model that allows one to estimate the cost, effort, and schedule when planning a new software development activity. COCOMO® II is the latest major extension to the original COCOMO® (COCOMO® 81) model published in 1981. It consists of three Sub models, each one offering increased fidelity the further along one is in the project planning and design process. Listed in increasing fidelity, these sub models are called the Applications Composition, Early Design, and Post-architecture models.

**COCOMO® II can be used for the following major decision situations**

- Making investment or other financial decisions involving a software development effort
- Setting project budgets and schedules as a basis for planning and control
- Deciding on or negotiating tradeoffs among software cost, schedule, functionality, performance or quality factors
- Making software cost and schedule risk management decisions
- Deciding which parts of a software system to develop, reuse, lease, or purchase
- Making legacy software inventory decisions: what parts to modify, phase out, outsource, etc

- Setting mixed investment strategies to improve organization's software capability, via reuse, tools, process maturity, outsourcing, etc

In the COCOMO II model, some of the most important factors contributing to a project's duration and

Costs are the Scale Drivers. You set each Scale Driver to describe your project; these Scale Drivers determine the exponent used in the Effort Equation.

The 5 Scale Drivers are:

- Precedentedness
- Development Flexibility
- Architecture / Risk Resolution
- Team Cohesion
- Process Maturity

Scale Drivers have replaced the Development Mode of COCOMO 81. The first two Scale Drivers, Precedentedness and Development Flexibility actually describe much the same influences that the original Development Mode did. In the COCOMO II model, some of the most important factors contributing to a project's duration and cost are the Scale Drivers. You set each Scale Driver to describe your project; these Scale Drivers determine the exponent used in the Effort Equation.

The 5 Scale Drivers are:

Note that the Scale Drivers have replaced the Development Mode of COCOMO 81. The first two Scale Drivers, Precedentedness and Development Flexibility actually describe much the same influences that the original Development Mode did.

### **Cost Drivers**

COCOMO II has 17 cost drivers. As you assess your project, development environment, and team to set each cost driver. The cost drivers are multiplicative factors that determine the effort required to complete the software project. For example, if your project will develop software that controls an airplane's flight, you would set the Required Software Reliability (RELY) cost driver to Very High. That rating corresponds to an effort multiplier of 1.26, meaning that your project will require 26% more effort than a typical software project.

COCOMO II defines each of the cost drivers, and the Effort Multiplier associated with each rating. Check the Costar help for details about the definitions and how to set the cost drivers.

### **COCOMO II Effort Equation**

The COCOMO II model makes its estimates of required effort (measured in Person-Months or PM) based primarily on your estimate of the software project's size (as measured in thousands of SLOC, KSLOC):

$$\text{Effort} = 2.94 * \text{EAF} * (\text{KSLOC})^E$$

Where

EAF Is the Effort Adjustment Factor derived from the Cost Drivers

E Is an exponent derived from the five Scale Drivers

As an example, a project with all Nominal Cost Drivers and Scale Drivers would have an EAF of 1.00 and exponent, E, of 1.0997. Assuming that the project is projected to consist of 8,000 source lines of code, COCOMO II estimates that 28.9 Person-Months of effort is required to complete it:

$$\text{Effort} = 2.94 * (1.0) * (8)^{1.0997} = 28.9 \text{ Person-Months}$$

### **Effort Adjustment Factor**

The Effort Adjustment Factor in the effort equation is simply the product of the effort multipliers corresponding to each of the cost drivers for your project.

For example, if your project is rated Very High for Complexity (effort multiplier of 1.34), and Low for Language & Tools Experience (effort multiplier of 1.09), and all of the other cost drivers are rated to be Nominal (effort multiplier of 1.00), the EAF is the product of 1.34 and 1.09.

$$\text{Effort Adjustment Factor} = \text{EAF} = 1.34 * 1.09 = 1.46$$

$$\text{Effort} = 2.94 * (1.46) * (8)^{1.0997} = 42.3 \text{ Person-Months}$$

### **COCOMO II Schedule Equation**

The COCOMO II schedule equation predicts the number of months required to complete your software project. The duration of a project is based on the effort predicted by the effort equation:

$$\text{Duration} = 3.67 * (\text{Effort})^{\text{SE}}$$

Where

Effort is the effort from the COCOMO II effort equation

SE Is the schedule equation exponent derived from the five Scale Drivers

Continuing the example, and substituting the exponent of 0.3179 that is calculated from the scale drivers, yields an estimate of just over a year, and an average staffing of between 3 and 4 people:

$$\text{Duration} = 3.67 * (42.3)^{0.3179} = 12.1 \text{ months}$$

$$\text{Average staffing} = (42.3 \text{ Person-Months}) / (12.1 \text{ Months}) = 3.5 \text{ people}$$

### **The SCED Cost Driver**

The COCOMO cost driver for Required Development Schedule (SCED) is unique, and requires a special explanation. The SCED cost driver is used to account for the observation that a project developed on an accelerated schedule will require more effort than a project developed on its optimum schedule. A SCED rating of Very Low corresponds to an Effort Multiplier of 1.43 (in the COCOMO II.2000 model) and means that you intend to finish your project in 75% of the optimum schedule (as determined by a previous COCOMO estimate). Continuing the example used earlier, but assuming that SCED has a rating of Very Low, COCOMO produces these estimates:

$$\text{Duration} = 75\% * 12.1 \text{ Months} = 9.1 \text{ Months}$$

**Effort Adjustment Factor = EAF =  $1.34 * 1.09 * 1.43 = 2.09$**

**Effort =  $2.94 * (2.09) * (8)1.0997 = 60.4$  Person-Months**

**Average staffing =  $(60.4 \text{ Person-Months}) / (9.1 \text{ Months}) = 6.7$  people**

Calculation of duration isn't based directly on the effort (number of Person-Months) i.e.  $\frac{1}{2}$  instead it's based on the schedule that would have been required for the project assuming it had been developed on the nominal schedule. Remember that the SCED cost driver means "accelerated from the nominal schedule". The Costar command Constraints | Constrain Project displays a dialog box that lets you trade off duration vs. effort (SCED is set for you automatically). You can use the dialog box to constrain your project to have a fixed duration, or a fixed cost.

**4b) Describe in detail COCOMO model for software cost estimation. Use it to estimate the effort required to build software for a simple ATM that produces 12 screens, 10 reports and 80 software components. Assume average complexity and average developer maturity. Use application composition model with object points. [Nov / Dec 2016]**

**Solution :**

The project Resources are :

1. Human Resource.
2. Reusable Software Resource.
3. Environmental Resource.

New object point , NOP =  $12 \times 2 + 20 \times 5 + 88 = 212$

Productivity Rate , PROD = 13

Therefore , Estimated Effort =  $NOP / PROD = 212 / 13 = 16.31$

## **SOFTWARE PROJECT PLANNING**

### **5. Explain Software Project Planning**

**May: 05, 06, Dec: 06, 07, May:15**

- Software project planning encompasses five major activities

Estimation, scheduling, risk analysis, quality management planning, and change management planning.

Estimation determines how much money, effort, resources, and time it will take to build a specific system or product

- The software team first estimates
  - The work to be done
  - The resources required
  - The time that will elapse from start to finish
- Then they establish a project schedule that

- Defines tasks and milestones
- Identifies who is responsible for conducting each task
- Specifies the inter-task dependencies
- Planning requires technical managers and the software team to make an initial commitment
- Process and project metrics can provide a historical perspective and valuable input for generation of quantitative estimates
- Past experience can aid greatly
- Estimation carries inherent risk, and this risk leads to uncertainty
- The availability of historical information has a strong influence on estimation risk

### **Task Set for Project Planning**

- 1) Establish project scope
- 2) Determine feasibility
- 3) Analyze risks
- 4) Define required resources
  - a) Determine human resources required
  - b) Define reusable software resources
  - c) Identify environmental resources
- 5) Estimate cost and effort
  - a) Decompose the problem
  - b) Develop two or more estimates using different approaches
  - c) Reconcile the estimates
- 6) Develop a project schedule
  - a) Establish a meaningful task set
  - b) Define a task network
  - c) Use scheduling tools to develop a timeline chart
- 7) Define schedule tracking mechanisms

S/w project management process begins with project planning objective of software project planning - to provide a framework for manager to make reasonable estimates of resources, costs and schedules

### **Activities associated with project planning**

- ☐ Software scope
- ☐ resources
- ☐ Project estimation
- ☐ Decomposition

### **Software Scope**

- ☐ want to establish a project scope that is unambiguous and understandable at management and technical levels
- ☐ describes:
  - o function
  - o performance
  - o constraints

- o interfaces
- o reliability

## Resources

- ☐ Must estimate resources required to accomplish the development effort
- ☐ Three major categories of software engineering resources
  - o People
  - o Development environment
  - o Reusable software components
- ☐ Often neglected during planning but become a paramount concern during the construction phase of the software process
- ☐ Each resource is specified with
  - o A description of the resource
  - o A statement of availability
  - o The time when the resource will be required
  - o The duration of time that the resource will be applied
- ☐ Off-the-shelf components
  - o Components are from a third party or were developed for a previous project
  - o Ready to use; fully validated and documented; virtually no risk
- ☐ Full-experience components
  - o Components are similar to the software that needs to be built
  - o Software team has full experience in the application area of these components
  - o Modification of components will incur relatively low risk
- ☐ Partial-experience components
  - o Components are related somehow to the software that needs to be built but will require substantial modification
  - o Software team has only limited experience in the application area of these components
  - o Modifications that are required have a fair degree of risk
- ☐ New components
  - o Components must be built from scratch by the software team specifically for the needs of the current project
  - o Software team has no practical experience in the application area
  - o Software development of components has a high degree of risk

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

## I)PROJECT SCHEDULING II) TIMELINE CHARTS

6. Write shot notes on i) Project Scheduling ii) Timeline Charts  
06,Dec:06,07,May:15

May: 05,

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. Schedule identifies all major process framework activities and the product functions to which they are applied. Scheduling for software engineering projects can be viewed from two rather different perspectives. I) first, an end date for release of a computer-based system. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization.

### 1 Basic Principle

A number of basic principles guide software project scheduling:

**Compartmentalization.** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.

**Interdependency.** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available.

**Time allocation.** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

**Effort validation.** Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time

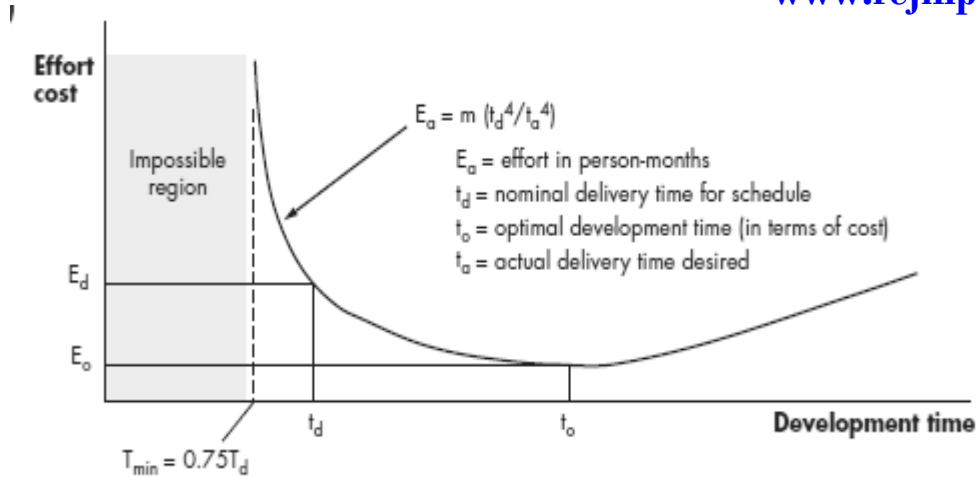
**Defined responsibilities.** Every task that is scheduled should be assigned to a specific team member.

**Defined outcomes.** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

**Defined milestones.** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 15) and has been approved.

### 2. The Relationship between People and Effort

In small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.



The curve indicates a minimum value  $t_o$  that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). The PNR curve also indicates that the lowest cost delivery option,  $t_o \approx 2t_d$ . The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay. The number of Delivered lines of code (source statements),  $L$ , is related to effort and development time by the equation:

$$L = P * E^{1/3} t^{4/3}$$

where  $E$  is development effort in person-months,  $P$  is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for  $P$  range between 2000 and 12,000), and  $t$  is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort  $E$ :

$$E = L^3 / p^3 t^4$$

Where  $E$  is the effort expended (in person-years) over the entire life cycle for software development and maintenance and  $t$  is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor (\$/person-year).

### Defining a task set for the software project

A task set is a collection of software engineering tasks, milestones and deliverables that must be accomplished to complete the project. A task network, called also activity network, is a graphic representation of the task flow of a project. It depicts the major software engineering tasks from the selected process model arranged sequentially or in parallel. Consider the task of developing a software library information system.

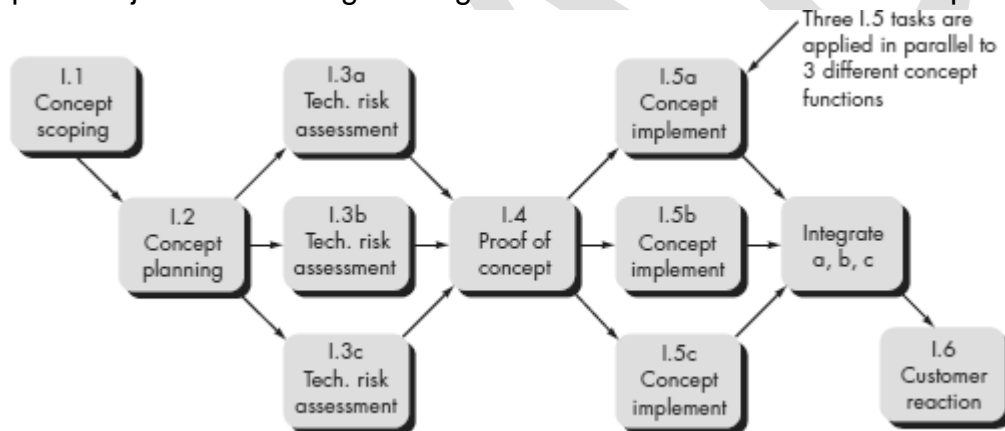
The scheduling of this system must account for the following requirements (the subtasks are given in *italic*): - initially the work should start with design of a control terminal ( $T_0$ ) class for no more than eleven working days; - next, the classes for student user ( $T_1$ ) and faculty user ( $T_2$ ) should be designed in parallel, assuming that

the elaboration of student user takes no more than six days, while the faculty user needs four days; -

when the design of student user completes there have to be developed the network protocol (T4), it is a subtask that requires eleven days, and simultaneously there have to be designed network management routines (T5) for up to seven days; - after the termination of the faculty user subtask, a library directory (T3) should be made for nine days to maintain information about the different users and their addresses; - the completion of the network protocol and management routines should be followed by design of the overall network control (T7) procedures for up to eight days; - the library directory design should be followed by a subtask elaboration of library staff (T6), which takes eleven days; - the software engineering process terminates with testing (T8) for no more than four days

### Defining a task network

A task network, also called an activity network, is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering actions. A task network for concept development



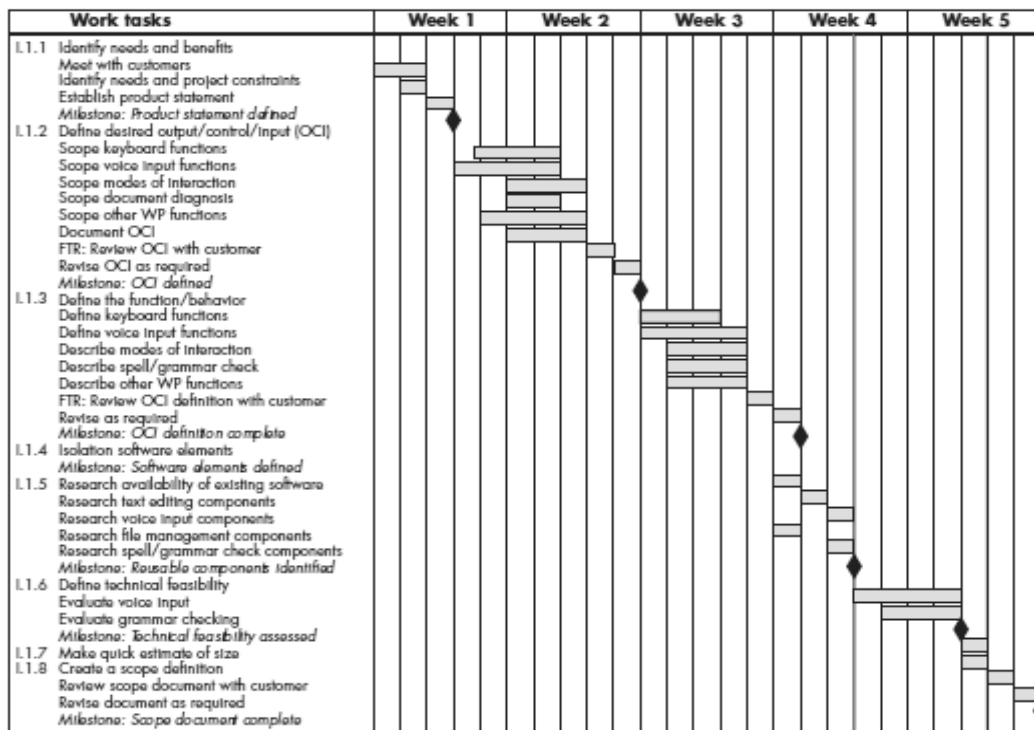
Program evaluation and review technique (PERT) and the critical path method (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected. Interdependencies among tasks may be defined using a task network. Tasks,

Sometimes called the project work breakdown structure (WBS), are defined for the product as a whole or for individual functions. Both PERT and CPM provide quantitative tools that allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish “most likely” time estimates for individual tasks by applying statistical models, and (3) calculate “boundary times” that define a time “window” for a particular task.

### Time-line chart

Time-line chart, also called a Gantt chart, is generated. A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual

Working on the project. Format of a time-line chart. It depicts a part of a software project schedule that emphasizes the concept scoping task for a word-processing (WP) software product. All project tasks are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate the milestones.



## Tracking the Schedule

Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems
- Evaluating the results of all reviews conducted throughout the software engineering process
- Determining whether formal project milestones (the diamonds shown in Figure) have been accomplished by the scheduled date
- Comparing the actual start date to the planned start date for each project task listed in the resource table
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon

## Earned value analysis

EVA is a technique of performing quantitative analysis of the software project. It provides a common value scale for every task of software project.

The EVA acts as a measure for software project progress.

To determine the earned value, the following steps are performed:

1. The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWS<sub>i</sub> is the effort planned for work task *i*. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWS<sub>i</sub> values for all work tasks that should have been completed by that point in time on the project schedule.

2. The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence,

$$BAC = \sum (BCWS_k) \text{ for all tasks } k$$

3. Next, the value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Schedule performance index,  $SPI = BCWP / BCWS$

Schedule variance,  $SV = BCWP - BCWS$

Percent complete  $\% = BCWP / BAC$

## Effort Distribution

A recommended distribution of effort across the software process is often referred to as the *40–20–40 rule*. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk.

Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered. Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated even higher percentages are typical.

## PROBLEM-BASED ESTIMATION

### 7. Write short notes on Problem-Based Estimation. May: 05, 06, 07, Dec-07, 10.

#### Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation:

- (1) As estimation variables to “size” each element of the software and
- (2) As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common.

LOC or FP (the estimation variable) is then estimated for each function. Function estimates are combined to produce an overall estimate for the entire project. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for past productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate. Using historical data or (when all else fails) intuition,

Estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. A three-point or expected value can then be computed.

The expected value for the estimation variable (size)  $S$  can be computed as a weighted average of the optimistic ( $sopt$ ), most likely ( $sm$ ), and pessimistic ( $spess$ ) estimates.

#### An Example of LOC-Based Estimation

Following the decomposition technique for LOC, an estimation table is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic, 4600 LOC; most likely, 6900 LOC; and pessimistic, 8600 LOC. Applying Equation the

expected value for the 3D geometric analysis functions is 6800 LOC. Other estimates are derived in a similar fashion.

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	<i>33,200</i>

By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system. A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of \$8000 per month, the cost per line of code is approximately \$13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is \$431,000 and the estimated effort is 54 person-months.<sup>7</sup>

### **An Example of FP-Based Estimation**

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the table we would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software. An FP value is computed using the technique

FP estimated \_ count total \_  $[0.65 \_ 0.01 \_ \_ (Fi)] \_ 375$  The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of \$8000 per month, the cost per FP is approximately \$1230. Based on the FP estimate and the historical productivity data, the total estimated Project cost is \$461,000 and the estimated effort is 58 person-months.

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	97
Number of external outputs	12	15	22	16	5	78
Number of external inquiries	16	22	28	22	5	88
Number of internal logical files	4	4	5	4	10	42
Number of external interface files	2	2	3	2	7	15
<i>Count total</i>						320

Each of the complexity weighting factors is estimated, and the value adjustment factor is computed as described in Chapter 23:

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
<b>Value adjustment factor</b>	<b>1.17</b>

Finally, the estimated number of FP is derived:

## PART – C

### 1. Explain in detail about Software Configuration Management (May/Jun 2014)

Software Configuration management is a set of activities carried out for identifying, organizing and controlling changes throughout the lifecycle of computer software.

During the development of software change must be managed and controlled in order to improve quality and reduce error. Hence Software Configuration Management is a quality assurance activity that is applied throughout the software process.

#### Need for SCM

The software configuration management is concerned with managing the changes in the evolving software. If the changes are not controlled at all then this stream of uncontrolled change can cause the well-running software project into chaos. Hence it is essential to perform following activities -

- Identify these changes
- Control the changes
- Ensure that the changes are properly implemented and
- Report these changes to others.

The software configuration management may be seen as part of quality management process.

#### Baseline

- The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:

A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

- A baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of them is obtained through formal technical review
- Further changes to the program architecture (which is actually documented in the design model) can be made only after each has been evaluated and approved.

#### Configuration Management Planning

Configuration planning defines the standards and procedures that should be used for configuration management. Initially configuration management standards must be set. These standards must be adapted to fit the requirements and to identify constraints.

The configuration management plan consists of -

- Definition of managed configuration items and schemes used for identifying the entities.
- Find out the targets that are responsible for configuration management procedures.

- Define configuration management policies which can be used for control and Version management.
- Specify the name of the tool used for configuration management along with its use.
- Describe configuration management database structure so that stored information can be maintained.
- 

### **Software Configuration Item Identification**

A Software Configuration Item (SCI) is information that is created as part of the software Engineering process.

#### **Examples of Software Configuration Items are**

##### **COMPUTER PROGRAMS**

- Source programs
- Executable programs

##### **DOCUMENTS DESCRIBING THE PROGRAMS**

- Technical manual
- User's manual

##### **DATA**

- Program components or functions
- External data
- File structure

For each type of item, there may be a large number of different individual items produced. For instance there may be many documents for a software specification

such as project plan, quality plan, test plan, design documents, programs, test reports, review reports. These SCI or items will be produced during the project, stored, retrieved, changed, stored again, and so on.

## **2. The process of Delphi method? Advantages and disadvantages (Nov/Dec 2013)**

### **Process:**

- The Delphi technique comprises several steps involving participants who may or may not meet face to face.
- The participants (or panel) might be employees of a specific company conducting a Delphi project or experts selected from the outside for their in-depth knowledge of a given academic discipline or manufacturing process.

### **Advantage:**

There are several advantages to the Delphi technique.

- One of the most significant is its versatility. The technique can be used in a wide range of environments, e.g., government planning, business and industry predictions, volunteer group decisions.

- Another important advantage lies in the area of expenses.
- For example, the Delphi technique saves corporations money in travel expenses.
- They do not have to gather participants from several points of the globe in one place to resolve a problem or predict the future, yet they still can generate relevant ideas from the people best suited to offer their expertise.
- This is particularly beneficial to multinational corporations, whose executives and key personnel may be based in cities as far apart as Melbourne, New York, Tokyo, Buenos Aires, and London.
- The technique also protects participants' anonymity. Thus, they feel better protected from criticism over their proposed solutions and from the pitfalls of "groupthink".

#### **Disadvantage**

- The Delphi technique is somewhat time consuming, which renders it ineffective when fast answers are needed.
- It might also be deficient in the degree of fully thought-out resolutions. People acting together in a group setting benefit from others' ideas. Thus, there might be more insightful and pragmatic resolutions to problems offered by people in interactive settings, e.g., through the nominal group technique,
- in which participants are gathered in one place but operate independently of one another.

### **3. Discuss Putnam resources allocation model .Derive the time and effort equations. May : 16**

The Putnam model is an empirical software effort estimation model. As a group, empirical models work by collecting software project data (for example, effort and size) and fitting a curve to the data. Future effort estimates are made by providing size and calculating the associated effort using the equation which fit the original data . It is one of the earliest of these types of models developed, and is among the most widely used. Closely related software parametric models are Constructive Cost Model (COCOMO).

#### **The Software Equation**

While managing R&D projects for the Army and later at GE, Putnam noticed software staffing profiles followed the well-known Rayleigh distribution.

Putnam used his observations about productivity levels to derive the software equation:

$$B^{1/3} \text{ size} / \text{Productivity} = \text{Effort}^{1/3} \cdot \text{Time}^{4/3}$$

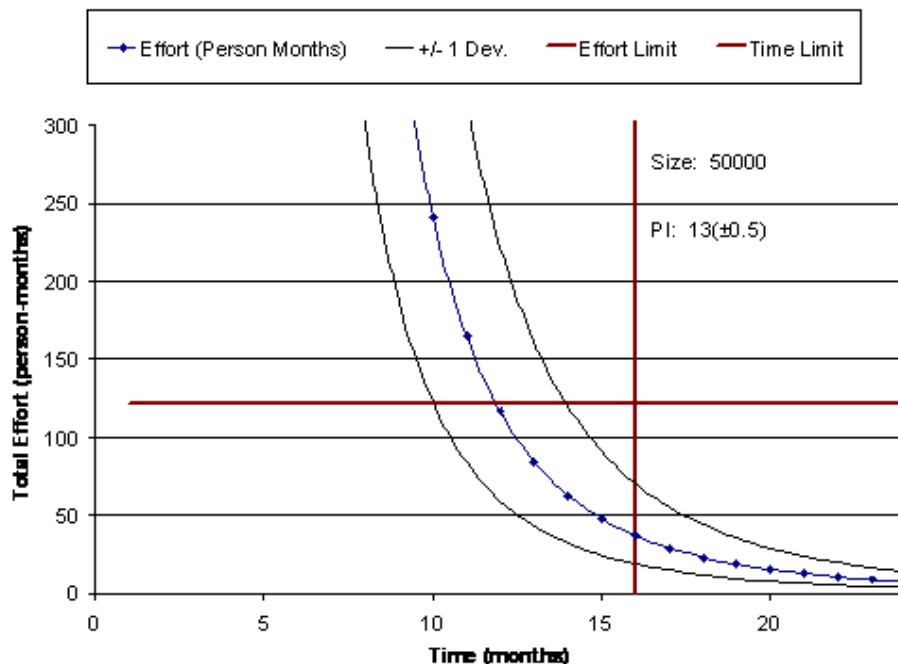
where:

- Size is the product size (whatever size estimate is used by your organization is appropriate). Putnam uses ESLOC (Effective Source Lines of Code) throughout his books.
- B is a scaling factor and is a function of the project size.
- Productivity is the Process Productivity, the ability of a particular software organization to produce software of a given size at a particular defect rate.
- Effort is the total effort applied to the project in person-years.
- Time is the total schedule of the project in years.

In practical use, when making an estimate for a software task the software equation is solved for effort:

$$\text{Effort} = [\text{size} / \text{Productivity} \cdot \text{Time}^{4/3}]^3 B$$

An estimated software size at project completion and organizational process productivity is used. Plotting effort as a function of time yields the Time-Effort Curve. The points along the curve represent the estimated total effort to complete the project at some time. One of the distinguishing features of the Putnam model is that total effort decreases as the time to complete the project is extended. This is normally represented in other parametric models with a schedule relaxation parameter.



This estimating method is fairly sensitive to uncertainty in both size and process productivity. Putnam advocates obtaining process productivity by calibration Putnam

makes a sharp distinction between 'conventional productivity' size / effort and process productivity.

One of the key advantages to this model is the simplicity with which it is calibrated. Most software organizations, regardless of maturity level can easily collect size, effort and duration (time) for past projects. Process Productivity, being exponential in nature is typically converted to a linear productivity index an organization can use to track their own changes in productivity and apply in future effort estimates

**4. Suppose you have a budgeted cost of a project as Rs. 9,00,000 . The project is to be completed in 9 months. After a month , you have completed 10 percent of the project at a total expense of Rs.1,00,000. The planned completion should have been 15 percent. You need to determine whether the project is on -time and on - budget? Use Earned value analysis approach and interpret.**

**[Nov / Dec 2016]**

**Solution :**

Here , BAC = \$ 900000

AC = \$ 100000

The planned value and Earned value can computed as,

- Planned Value = Planned Completion (%) x BAC  
= 15 % x \$ 900000  
= \$ 135,000
- Earned Value = Actual Completion (%) x BAC  
= 10 % x \$ 900000  
= \$ 90,000

Compute the earned value Variances :

- Cost Performance Index (CPI) = EV / AC  
= \$ 90,000 / \$ 100,000 = 0.90

This means for every \$1 spent, the project is producing only 90 cents in work.

- Schedule Performance Index (SPI) = EV / PV  
= \$90,000 / \$135,000 = 0.67

This means for every estimated hour of work ,the project team is completing only 0.67 hours (approximately 40 minutes).

**Interpretation:** Since both cost performance index (CPI index) and schedule performance index (SPI Index) are less than 1. It means that the project is over budget and behind schedule. This example project is in major trouble and corrective action needs to be taken. Risks management needs to kick-in.

5. An application has the following : 10 low external inputs, 8 high external outputs, 13 low internal logical files, 17 high external interface files, 11 average external inquires and complexity adjustments factor of 1.10. What are the unadjusted and adjusted function point counts? [May / june 2016]

Information Domain value	Count		Weighting factor		
			Simple	Average	Complex
Low external inputs (LET's)	10	x	3	4	6=30
High External outputs (HEO's)	8	x	4	5	7=32
Low internal logical files	13	x	3	4	6=39
High External (HEIF) Interface files	17	x	7	10	15=119
Average external inquires	11	x	5	7	10=5
				count	275

Adjustment Factor =1.10

$$=275 \times [0.65 + (0.01 \times 46)]$$

$$=305.25$$

$$=305$$

## INDUSTRY CONNECTIVITY AND LATEST DEVELOPMENTS

### 1. Agile Software development.

Agile methods break tasks into small increments with minimal planning and do not directly involve long-term planning. Iterations are short time frames (time boxes) that typically last from one to four weeks. Each iteration involves a team working through a full software development cycle, including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. This minimizes overall risk and allows the project to adapt to changes quickly. Stakeholders produce documentation as required. Multiple iterations might be required to release a product or new features.

### 2. Real Analytics

Organizations began to see information automation outweigh business process automation as their highest priority area. Analytics offer improved visibility to drive operational efficiencies, as well as a platform for growth by addressing heart-of-the-business questions that could guide decisions, yield new insights and help predict the future of business. Leading organizations are launching broad initiatives with executive-level sponsorship, ready and eager to achieve their vision via real analytics. Due to exploding data volumes and regulators demands deeper insight into risks, exposure and public responsiveness are much needed.

### 3. Applied Mobility: The Mobile applications are essentially powerful that they are elegant solutions to well-defined problems, and designed for operations on-the-go. Companies are rethinking business processes and enabling new business models that would not have been possible without mobile technology. Evolutions in location-based services, social networks, mobile payment processing, low-cost device add-ons and integration with enterprise systems has led to the potential for employees, customers and suppliers to consume and produce sophisticated information, goods and services from anywhere. And with the extension of mobile solutions to sensors and actuators in physical goods and equipment, Capability Clouds

### 4. Capability clouds

Clouds move beyond the building blocks of capacity to deliver finished services that directly address business objectives and enterprise goals. Instead of dealing with machine images or database instances, the discussion shifts to the analytics cloud, the testing cloud or the sales cloud and help to focus on a more important set of values.. It is relatively easy for a business unit leader to buy a software-as-a-service tool for point solutions such as workforce planning or compensation management

### 5. Social Computing

Everyone has started going online for their daily needs and hence we are leaving the trails our opinion, behavior and choices. The data of traces created when mined would provide business with a good source of insight on market positioning, consumer sentiment and employee productivity. By performing analytical operation on the data organizations can better understand their customer needs, preferences, their employee's experiences and problems that require immediate co-corporate attention.

CS6403 –SOFTWARE ENGINEERING

(Regulation 2013)

Answer ALL questions

PART- A (10 x 2 =20)

1. What led to the transition from product oriented development to process oriented development? [Page.No:6]
2. Mention the characteristics of software contrasting it with characteristics of hardware. [Page.No:6]
3. List the good characteristics of a good SRS. [Page.No:40]
4. What are the linkages between data flow and E-R diagram? [Page.No:38]
5. If a module has logical cohesion, what kind of coupling is this module likely to have? [Page.No:77]
6. What is the need for architectural mapping using data flow? [Page.No:77]
7. How can refactoring be made more effective? [Page.No:110]
8. Why does software fail it has passed from acceptance testing? [Page.No:110]
9. List a few process and product metrics. [Page.No:145]
10. Will exhaustive testing guarantee that the program is 100% correct? [Page.No:146]

**PART – B (5x16=80 Marks )**

- 11.(a) (i) Discuss the prototyping model.What is the effect of designing a prototype on the overall cost of software project? (8)[pg.no:07]  
(ii) Describe the types of situations where iterative enhancement model might lead to difficulties. (8) [pg.no:07]  
OR  
(b)(i) Elucidate the key features of the software process models with suitable examples. (8) [pg.no:07]  
(ii) What is the role of user participation in the selection of a lifecycle model? (8) [pg.no:07]
- 12.(a)(i) Explain the organizations of SRS and highlight the importance of each subsection.(8) [pg.no:45]  
(ii) Requirements analysis is unquestionably the most communication intensive step in the software engineering process.Why does the communication for path frequently breaks down? (8)[pg.no:58]  
OR  
(b)(i) Differentiate between user and system requirements. (4) [pg.no:40]

(ii) Describe the requirements change management process in detail.(12)

[pg.no:52]

13.(a)Write short notes on the following. (4x4=16)

[pg.no:96,101]

- (i)Design heuristics
- (ii)User-interface design
- (iii)Component level design
- (iv)Data/Class design

OR

(b)(i) What is modularity?State its importance and explain coupling and cohesion. (8)

[pg.no:94]

(ii) Discuss the differences between object oriented and function oriented design. (8)

[pg.no:107]

14.(a)(i) State the need for refactoring.How can a development model benefit by the use of refactoring? (8)

[pg.no:134]

(ii) Why does software testing need extensive planning?Explain.

(8)[pg.no:125]

OR

(b)(i) Compare and contrast alpha and beta testing. (8)

[pg.no:143]

(ii) Consider a program for determining the previous date. Its input is a triple of day ,month and year with the values in the range  $1 \leq \text{month} \leq 12$  ,  $1 \leq \text{day} \leq 31$  ,  $1990 \leq \text{year} \leq 2014$ . The possible outputs would be previous date or invalid input date. Design the boundary value test cases. (8)

[pg.no:141]

15. (a) Write short notes on the following (2x8=16)

[pg.no:158]

- I. Make / Buy Decision
- II. COCOMO II

OR

(b) (i) An application has the following : 10 low external inputs, 8 high external outputs,13 low internal logical files,17 high external interface files, 11 average external inquires and complexity adjustments factor of 1.10. What are the unadjusted and adjusted function point counts?(4)

[pg.no:177]

(ii) Discuss Putnam resources allocation model .Derive the time and effort equations. (12)

[pg.no:174]

CS6403 –SOFTWARE ENGINEERING

(Regulation 2013)

Answer ALL questions

PART- A (10 x 2 =20)

1. If you have to develop a word processing software product ,what process model will use you choose? Justify your answer. [pg.no:6]
2. Depict the relationship between work product task activity and system. [pg.no:6]
3. Classify the following as functional / non functional requirements for a banking system. [pg.no:40]
  - a. Verifying bank balance.
  - b. Withdrawing money from bank.
  - c. Completion of transactions in less than one second.
  - d. Extending the system by providing more tellers for customers.
4. What is data dictionary? [pg.no:38]
5. What architectural styles are preferred for the following ? why ? [pg.no:76]
  - a) Net working
  - b) Web based Systems
  - C)Banking System
6. What UI design patterns are used for the following? [pg.no:76]
  - a. Page layout.
  - b)Tables.
  - C)Navigation through menus and web pages.
  - d. Shopping cart.
7. What methods are used for breaking very long expression and statements? [pg.no:110]
8. What is the difference between verification and validation? Which types of testing address verification? Which types of testing address validation? [pg.no:108]
9. What is risk management? [pg.no:146]
10. How is productivity and cost related to function points? [pg.no:144]

**PART – B (5x16=80 Marks )**

- 11.(a) Which process model is best suited for risk management?Dicuss in detail with an example. Give the advantage and disadvantages of the model. (16) [pg.no:07].

OR

- (b)(i) List the principles of agile software development.(8) [pg.no:35]

- (ii) Consider 7 functions with their estimated lines of code below. (8)

Function	LOC
Func1	2340

Func2	5380
Func3	6800
Func4	3350
Func5	4950
Func6	2140
Func6	8400

Average productivity based on historical data is 620 LOC/pm and labour rate is Rs.8000 per month. Find the total estimated project cost and effort. [pg.no:23]

12. (a) what is requirement elicitation? Briefly described the various activities performed in requirements elicitation phase with an example of the watch system that facilitates to set time and alarm. (16) [pg.no:53]

OR

- (b) What is the purpose of data flow diagrams? What are the notations used for the same. Explain by constructing a context flow diagram level -0 DFD and level-1 DFD for a library management system. [pg.no:60]

13. (a) What is structured design? Illustrate the structured design process from DFD to structured chart with a case study. (16) [pg.no:87]

OR

- (b)(i) Describe the golden rules for interface design. (8) [pg.no:96]

- (ii) Explain component level design with suitable examples. (8) [pg.no:105]

14. (a)(i) Consider the pseudocode for simple subtraction given below : [10]

(1) Program 'Simple Subtraction'

(2) Input (x,y)

(3) Output (x)

(4) Output (y)

(5) If  $x > y$  then DO

(6)  $x - y = z$

(7) Else  $y - x = z$

(8) EndIf

(9) Output (z)

(10) Output " End Program"

Perform basis path testing and generate test cases. [pg.no:130]

- (ii) What is refactoring? When is it needed? Explain with an example.

[pg.no:134]

OR

(b) What is black box testing? Explain the different types of black box testing strategies. Explain by considering suitable examples. [pg.no:114]

15.(a)(i) Suppose you have a budgeted cost of a project as Rs. 9,00,000 . The project is to be completed in 9 months. After a month , you have completed 10 percent of the project at a total expense of Rs.1,00,000. The planned completion should have been 15 percent. You need to determine whether the project is on - time and on - budget? Use Earned value analysis approach and interpret.(8)

[pg.no:176]

(ii) Consider the following function point components and their complexity. If the total degree of influence is 52, find the estimated function points. (8)

[pg.no:157]

Function Type	Estimated Count	Complexity
ELF	2	7
ILF	4	10
EQ	22	4
EO	16	5
EI	24	4

OR

(b) Describe in detail COCOMO model for software cost estimation. Use it to estimate the effort required to build software for a simple ATM that produces 12 screens, 10 reports and 80 software components. Assume average complexity and average developer maturity. Use application composition model with object points. (16) [pg.no:161]

**B.E. /B. Tech DEGREE EXAMINATION, APRIL/MAY 2015**

**Fourth Semester**

**Computer Science and Engineering  
CS6403 – SOFTWARE ENGINEERING  
(Common to Information Technology)  
(Regulation 2013)**

**Time: Three hours**

**Maximum: 100marks**

**Answer ALL questions.**

**PART A – (10 X 2 =20 MARKS)**

1. Write the Process framework and Umbrella activities. Pg.No : 4
2. State the advantages and disadvantages in LOC based Cost Estimation. Pg.No : 4
3. What is the need for feasibility analysis? Pg.No : 31
4. How are the requirements validated? Pg.No : 32
5. Draw diagrams to demonstrate the architectural styles. Pg.No : 65
6. List down the steps to be followed for User Interface design. Pg.No : 64
7. What is the need for regression testing? Pg.No : 93
8. Write the best practices for “CODING”. Pg.No : 92
9. Highlight the activities in Project planning. Pg.No : 122
10. State the importance of scheduling activity in project management. Pg.No : 123

**PART B – (5 X 16 = marks)**

11. (a). Neatly explain the following process models and write their advantages and disadvantages.
  - (i). Spiral model Pg.No : 6 (8)
  - (ii). Rapid Application Development model (8)Or  
(b). Discuss about the COCOMO models (Basic, Intermediate and detailed) for cost estimation. Pg.No. : 13 (16)
12. (a). Write about the following Requirements Engineering activities. Pg.No : 42
  - (i). Inception (2)
  - (ii). Elicitation (3)
  - (iii). Elaboration (3)
  - (iv). Negotiation (2)
  - (v). Specification (2)
  - (vi). Validation (2)
  - (vii). Requirements management (2)Or  
(b). Draw Use Case and Data Flow diagrams for a Restaurant System. The activities of Restaurant system are listed below.

Receive the Customer food Orders, Produce the customers ordered foods, Serve the customer with their ordered foods, Collect Payment from customers, Store customer payment details, Order Raw Materials for food products, Pay for Raw Materials and Pay for labor. Pg,Np :59

(16)

13. (a). Explain the various coupling and cohesion models used in Software design.  
Pg.No :81 (16)

Or

(b). For a case study of your choice show the architectural and Component design. Pg :68 (16)

14. (a). Describe the various black box and White box testing techniques. Use Suitable examples for your explanation. Pg.No:93,97  
(16)

Or

(b). Discuss about the various Integration and Debugging strategies followed in Software development. Pg.No :112 (16)

15. (a). State the need for Risk Management and explain the activities under Risk Management. Pg.No :24 (16)

Or

(b). Write short notes on the following. Pg.No : 138

(i). Project Scheduling. (8)

(ii). Project Timeline chart and Task network. (8)

**B.E/ B.TECH DEGREE EXAMINATION, MAY/JUNE 2014**

**Fifth Semester**

**Computer Science and Engineering**

**CS2301/CS 51/ 10144 CS 502- SOFTWARE ENGINEERING**

**(Regulation 2008/2010)**

**(Common to PTCS 2301- software engineering for B.E(part-time) Fifth semester computer science and engineering- regulation 2009)**

Time: Three hours

maximum: 100 marks

**Answer all questions**

**PART A-(10 x 2=20 marks)**

1. Distinguish verification and validation'
2. Define system engineering
3. What do you mean by functional and non-functional requirements?
4. List two advantages of employing prototyping in software process.
5. Define Software Architecture.
6. List the notations used in Data-flow models.
7. What are the classes of loops that can be tested?
8. What is Cyclomatic complexity?
9. What is software configuration management?
10. What is error tracking?

**PART B – (5 x 16 =80 marks)**

11. (a) Discuss in detail about any two evolutionary process models. (16)

**Or**

- (b)(i) Discuss about the classic waterfall process model. (8)  
(ii) Explain the prototype paradigm in process models. (8)

12. (a)(i) What are the components of the standard structure for the software requirements document? Explain in detail. (8)

- (ii) Write the software requirement specification for a system for your Choice . (8)

**Or**

- (b) What are the types of behavioral models ? Explain with examples.

13. (a) Explain in detail about four architectural styles. (16)

**Or**

- (b)(i) What are the Characteristics of a real time system? Explain why real time systems usually have to be implemented using concurrent process (10)

- (ii) Illustrate with the aid of an appropriate example how to design a real time monitoring and control systems. (6)

14. (a) Explain in detail about Integration Testing . (16)

**Or**

- (b) Explain in detail about Basic path testing (16)

15. (a)(i) What is COCOMO model? Explain in detail. (8)

(ii) What are CASE tools? Explain in the role of CASE tools in software development process (8)

**Or**

(b)(i) Elaborate on software Configuration Management (10)

(ii) What are the categories of software risks? Give an overview about risk Management. (6)

SCAD



- (b) A Coffee Vending Machine dispenses coffee to customers. Customers order coffee by selecting a recipe from a set of recipes. Customers pay for the coffee using coins. Change is given back, if any, to the customers. The 'Service Assistant' loads ingredients (coffee powder, milk, sugar, water, chocolate) into the coffee machine. The 'Service Assistant' adds a recipe by indicating the name of the coffee, the units of coffee powder, milk, sugar, water and chocolate to be added as well as the cost of the coffee. The Service Assistant can also edit and delete a recipe.

- (i) Develop the use case diagram for the specification above. (6)
- (ii) For any two scenarios draw an activity diagram and sequence diagram. (5 + 5)

12. (a) (i) What is the purpose of feasibility study? (2)
- (ii) State the inputs and results of the feasibility study. (4)
- (iii) List any four issues addressed by a feasibility study. (4)
- (iv) Elaborate the phases involved when carrying out a feasibility study. (6)

Or

- (b) (i) Differentiate functional and non-functional requirements. (5)
- (ii) For the requirement given below, identify stakeholders, functional and non-functional requirements: (11)

A software is to be built that will control an Automated Teller Machine (ATM). The ATM machine services customers 24 X 7. ATM has a magnetic stripe reader for reading an ATM card, a keyboard and display for interaction with the customer, a slot for depositing envelopes, a dispenser for cash, a printer for printing receipts and a switch that allows an operator to start/stop a machine.

The ATM services one customer at a time. When a customer inserts an ATM card and enters the personal identification number (PIN), the details are validated for each transaction. A customer can perform one or more transactions. Transactions made against each account are recorded so as to ensure validity of transactions.

If PIN is invalid, customer is required to re-enter PIN before making a transaction. If customer is unable to successfully enter PIN after three tries, card is retained by machine and customer has to contact bank.

The ATM provides the following services to the customer:

- (1) Withdraw cash in multiples of 100.
- (2) Deposit cash in multiples of 100.
- (3) Transfer amount between any two accounts.
- (4) Make balance enquiry.
- (5) Print receipt.

Each of the above transactions must be made within 60 seconds. In case the time exceeds 60 seconds, then the transaction is cancelled automatically. Also, if the machine is not used for more than two minutes after entry of card, the card is retained by the machine.

An operator panel with a key-operated switch allows an operator to start and stop the servicing of customers. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc. The operator is required to verify and enter the total cash on hand before starting the system from this panel.

13. (a) What is system modelling? Explain the process of creating models and the factors that should be considered when building models. (16)

Or

- (b) Tamil Nadu Electricity Board (TNEB) would like to automate its billing process. Customers apply for a connection (domestic/commercial). EB staff take readings and update the system. Each customer is required to pay charges bi-monthly according to the rates set for the type of connection. Customers can choose to pay either by cash/card. A bill is generated on payment. Monthly reports are provided to the EB Manager.

- (i) Give a name for the system. (1)  
 (ii) Draw the Level - 0 DFD (Context Flow Diagram) (5)  
 (iii) Draw the Level - DFD. (10)

14. (a) Given a set of numbers 'n', the function FindPrime(a[ ],n) prints a number if it is a prime number. Draw a control flow graph, calculate the cyclomatic complexity and enumerate all paths. State how many test cases are needed to adequately cover the code in terms of branches, decisions and statement? Develop the necessary test cases using sample values for 'a' and 'n'. (16)

Or

- (b) (i) What is black box testing? (2)  
 (ii) What is Equivalence Class Partitioning? list rules used to define valid and invalid equivalence classes. Explain the technique using examples. (7)  
 (iii) What is Boundary Value Analysis? Explain the technique specifying rules and its usage with the help of an example. (7)

15. (a) (i) Explain the COCOMO model for estimation. (8)  
 (ii) State ZIPF's law. (2)  
 (iii) What is the purpose of Delphi method? State advantages and disadvantages of the method. (6)

Or

- (b) What is Software Configuration Management? Explain various aspects of Configuration Management. (16)

**B.E/B.TECH DEGREE EXAMINATION, MAY/JUNE 2013**

**Fifth Semester**

**Computer Science and Engineering**

**CS2301/CS 51/10144 CS 502- SOFTWARE ENGINEERING**

**(Regulation 2008/2010)**

**Time: Three Hours**

**Maximum: 100 marks**

**Answer ALL Questions**

**PART A – (10x2=20Marks)**

1. What is software process? List its activities.
2. Distinguish between verification and validation.
3. List the benefits of software prototyping.
4. What is data dictionary?
5. Define data abstraction.
6. Write the use of data acquisition system.
7. What is black box testing?
8. Write down the generic characteristics of software testing.
9. Compare size-oriented with function oriented metrics.
10. What do you mean by estimation risk?

**PART-B (5x16=80 marks)**

11.(a)(i) With the neat sketch, explain the function of system engineering process (10)

(ii) What is computer based system? Explain the various elements used in it. (6)

Or

(b)(i) Explain the business process engineering hierarchy with an example (14)

(ii) What is the goal of product engineering? (2)

12.(a)(i) Explain the metrics used for specifying non – Functional requirements. (8)

(ii) Show the template of IEEE standard software Requirements document. (8)

Or

(b)(i) Explain the function of requirements engineering Process. (8)

(ii) Describe the use of behavioral model with Examples (8)

13. (a)(i) Discuss the design heuristics for effective Modularity design (8)

(ii) Explain the architectural styles used in Architectural design (8)

Or

(b)(i) List the activities of user interface design process (8)

(ii) Explain the general model of a real time system (8)

14. (a)(i) Explain the integration testing in detail (16)

Or

(b)(i) Write note on unit testing (8)

(ii) Explain the categories of debugging approaches (8)

15. (a)(i) Explain the use of COCOMO model (8)

(ii) Describe the steps involved in project scheduling Process. (8)

Or

(b)(i) Briefly discuss the activities of Software Configuration management. (8)

(ii) Explain the types of software project plan. (8)