TED (10)-4072 Reg. No. …….

(REVISION-2010) Signature ……..……

FIFTH SEMESTER DIPLOMA EXAMINATION IN COMPUTER ENGINEERING AND INFORMATIONTECHNOLIGY- OCTOBER, 2012

SOFTWARE ENGINEERING

[*Time:* 3 hours

(Maximum marks: 100)

Marks

PART –A
(Maximum marks: 10)

I. Answer all questions in two sentences. Each question carries 2 marks.

1. List any two qualities of a good Software Engineer

   - ❖ Good technical knowledge of the project areas (Domain knowledge).
   - ❖ Good communication skills. These skills comprise of oral, written, and interpersonal skills.

2. Define cohesion

   Cohesion is a measure of functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction between the two modules.

3. Write the need of driver module.

   A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

4. List any two reliability metrics

   - Rate of occurrence of failure (ROCOF)
   - Mean Time To Failure (MTTF)

5. Define data dictionary interface.

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

PART – B

II. Answer *any five* questions. Each question carries 6 marks

1. Explain project planning activities.

- Project planning

  Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

  1. Estimating the following attributes of the project:
     - Project size
     - Cost
     - Duration
     - Effort

  The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

  2. Scheduling: After the estimations are made, the schedules for manpower and other resources have to be developed.
  3. Staffing: staff organization and staffing plans have to be made.
  4. Risk Management: Risk identification, analysis, and abatement planning have to be done.
  5. Miscellaneous plans: Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

- Precedence ordering among project planning activities:

  Different project related estimates done by a project manager have already been discussed. Fig. shows the order in which important project planning activities may be undertaken. From fig. it can be easily observed that size estimation is the first

activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.

Fig. 11.1: Precedence ordering among planning activities

- Sliding Window Planning:

  Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However, project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning.

2. Describe different COCOMO models.

   COCOMO

   COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

   - Basic COCOMO Model:

     The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a1 x (KLOC)^a2 PM

Tdev=b1x (Effort)^b2 Months

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a1, a2, b1, b2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

Estimation of development effort:-

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : Effort = 2.4(KLOC)^1.05 PM

Semi-detached : Effort = 3.0(KLOC)^1.12 PM

Embedded : Effort = 3.6(KLOC)^1.20 PM

Estimation of development time:-

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = 2.5(Effort)^0.38 Months

Semi-detached :Tdev = 2.5(Effort)^0.35 Months

Embedded : Tdev = 2.5(Effort)^0.32 Months

- Intermediate COCOMO model

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development.

- Complete COCOMO model

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub- systems may have widely different characteristics.

The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

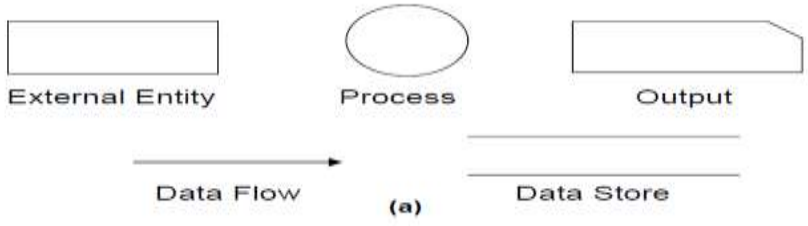
- Database part
- Graphical User Interface (GUI) part
- Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

3. Summarize DFD and its symbols.

- Data Flow Diagram (DFD)

The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols [as shown in fig.(a)] to represent the functions performed by a system and the data flow among these functions.



- Functional symbol:

A function represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions.

- External entity symbol:

  An external entity such as librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system inputting data to the system or by consuming the data produced by the system.

- Data flow symbol:

  A directed arc(or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.

- Data store symbol:

  A data store symbol is represented using two parallel lines. It represents a logical file. That is a, data store symbol can represent either a data structure or a physical fie on disk.

- Output symbol:

  The output symbol is used when a hard copy is produced.

4. List and explain the characteristics of a good software design.

   Characteristics of a good software design

   The definition of "a good software design" can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. The characteristics are listed below:

   - Correctness:- A good design should correctly implement all the functionalities identified in the SRS document.
   - Understandability:- A good design is easily understandable.

- Efficiency:- It should be efficient.
- Maintainability:- It should be easily amenable to change.

  Possibly the most important goodness criterion is design correctness. A design has to be correct to be acceptable. Given that a design solution is correct, understandability of a design is possibly the most important issue to be considered while judging the goodness of a design. A design that is easy to understand is also easy to develop, maintain and change. Thus, unless a design is easily understandable, it would require tremendous effort to implement and maintain it.

5. Distinguish between Internal and External software documentation.

   • Internal documentation

   Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code several forms. The important types of internal documentation are the following.

   - ❖ Comments embedded in the source code
   - ❖ Use of meaningful variable names
   - ❖ Module and function headers
   - ❖ Code indentation
   - ❖ Code structuring
   - ❖ Use of enumerated types
   - ❖ Use of constant identifiers
   - ❖ Use of use-defined data types

   - External documentation

   External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

   An important feature of good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the product. In other words all the documents developed for a product should be up-to-date, and every change made to the code should be reflected in the

relevant external documents. Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed.

6. Draw the CFG for sequence selection and iteration process.

Control Flow Graph (CFG):-

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig. a). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

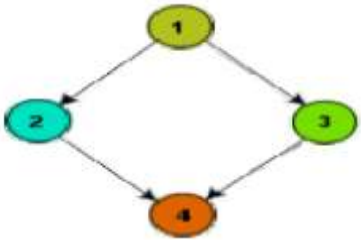
Fig. summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop.

SEQUENCE

1. a=5
2. b=a*2-1

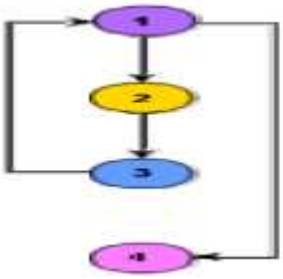
SELECTION

1. if (a>b)
2. c=3
3. else c=5
4. c=c*c.



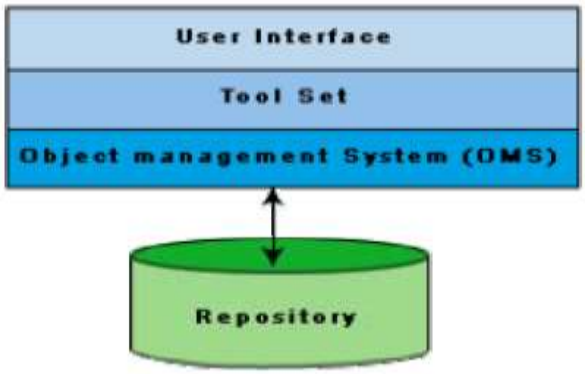
ITERATION

1. while(a>b){
2. b=b-1;
3. b=b*a;}
4. c=a+b;



7. Explain the architecture of modern CASE environment with a diagram.
   - Architecture of a CASE environment

     The architecture of a typical modern CASE environment is shown diagrammatically in fig. 15.2. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository.



     - User interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

- Object-Management System(OMS) and repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities such into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus the object management system takes care of appropriately mapping into the underlying storage management system.

PART – C

(Answer *one* full question from each unit. Each question carries 15 marks)

UNIT – I

III. (a) Explain content of Software Project Management Plan. (SPMP).

Software Project Management Plan (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below.

Organization of the Software Project Management Plan (SPMP) Document

1. Introduction
    a) Objectives
    b) Major Functions
    c) Performance Issues
    d) Management and Technical Constraints
2. Project Estimates
    a) Historical Data Used
    b) Estimation Techniques Used

c) Effort, Resource, Cost, and Project Duration Estimates

3. Schedule
    a) Work Breakdown Structure
    b) Task Network Representation
    c) Gantt Chart Representation
    d) PERT Chart Representation
4. Project Resources
    a) People
    b) Hardware and Software
    c) Special Resources
5. Staff Organization
    a) Team Structure
    b) Management Reporting
6. Risk Management Plan
    a) Risk Analysis
    b) Risk Identification
    c) Risk Estimation
    d) Risk Abatement Procedures
7. Project Tracking and Control Plan
8. Miscellaneous Plans
    a) Process Tailoring
    b) Quality Assurance Plan
    c) Configuration Management Plan
    d) Validation and Verification
    e) System Testing Plan
    f) Delivery, Installation, and Maintenance Plan

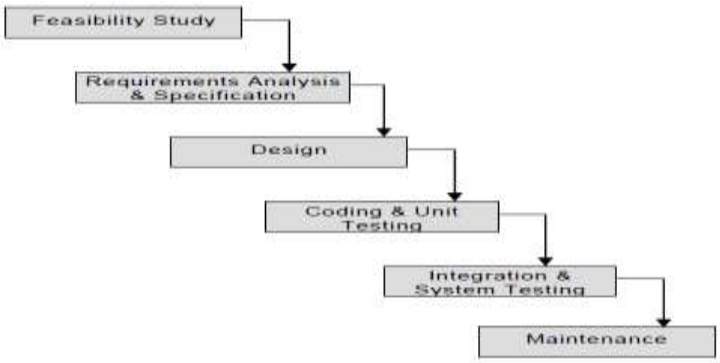
(b) Illustrate waterfall method with neat sketch.

❖ Classical Waterfall Model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it can't be used in actual software development projects.

Thus, this model can be considered to be a theoretical way of developing software. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.

Classical waterfall model divides the life cycle into the following phases:-



- Feasibility Study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

- Requirements Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

  - Requirements gathering and analysis, and
  - Requirements specification

- Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

- Coding and Unit Testing

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate

the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

- Integration and System Testing

  Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. Integration is normally carried out incrementally over a number of steps. α – testing: It is the system testing performed by the development team.

  - β – testing: It is the system testing performed by a friendly set of customers.
  - acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

- Maintenance

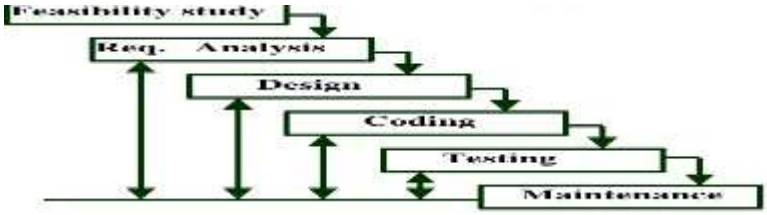
  Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:

  - Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
  - Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
  - Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

❖ Iterative Waterfall Model

The classical waterfall model is not a practical model in the sense that it can't be used in actual software development projects. Thus, this model can be considered to be a theoretical way of developing software. The iterative waterfall model as making

necessary changes to the classical waterfall model so that it becomes applicable to practical software development projects. Essentially the main change to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases as shown in fig.



OR

IV.

a) Explain LOC and give its merits and demerits.

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored. Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort

needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program.

- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size
- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities.

  It is very difficult to accurately estimate LOC in the final product from the problem specification.

b) Explain the spiral life cycle model.

- Spiral life cycle model

The Spiral model of software development is shown in fig. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. The following activities are carried out during each phase of a spiral model.

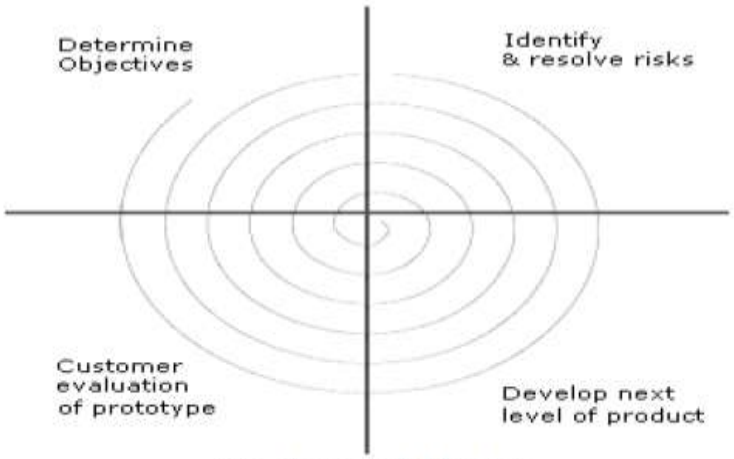


Fig. 2.2: Spiral Model

- ❖ First quadrant (Objective Setting):

- During the first quadrant, it is needed to identify the objectives of the phase.
- Examine the risks associated with these objectives.

- ❖ Second Quadrant (Risk Assessment and Reduction):
  - A detailed analysis is carried out for each identified project risk.
  - Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
- ❖ Third Quadrant (Development and Validation):
  - Develop and validate the next level of the product after resolving the identified risks.
- ❖ Fourth Quadrant (Review and Planning):
  - Review the results achieved so far with the customer and plan the next iteration around the spiral.
  - Progressively more complete version of the software gets built with each iteration around the spiral.

UNIT – II

V. List the contents of a good SRS and prepare an SRS document for the following software product. Implement a banking system having the provision to Add, Delete, Modify customers and to perform transactions like deposit and withdrawal(only described the contents)

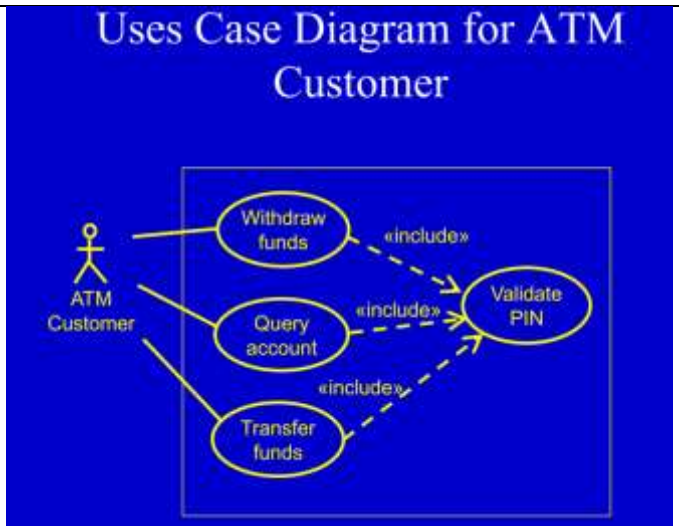
The benefits of a good SRS are,

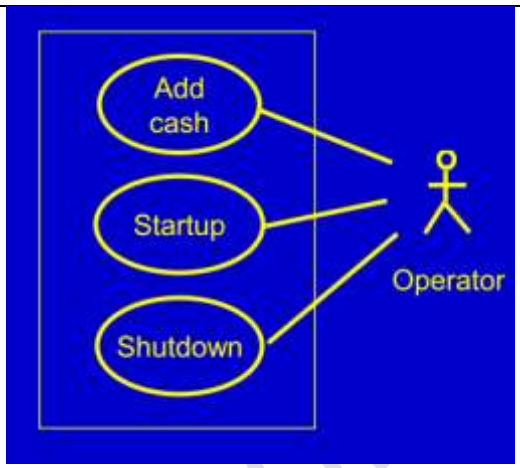
- A contract between the customer and the software vendor – A good SRS document specifies all the features required in the final system including technical requirements and interface requirements. SRS document is used by the customer to determine whether the software vendor has provided all the features in the delivered software system. To the Software vendor it provides a solid foundation to fix the scope of the software system.
- Enables costing and pricing of the project – A well defined SRS enables software developers to accurately estimate the amount of effort required to build the software product. Function point analysis and SMC are some the techniques adopted for estimating effort.
- Input for detailed design – A good SRS enables experienced developers to convert the requirements directly to a technical design. For example, a well defined data dictionary can be easily converted to a database specification.
- Management of customer expectations – Since SRS precisely defines project scope, it ensures that customer expectations don't change during software development. If they do, SRS can be modified and costing/pricing can be done again on the changes required.

A bank has several automated teller machines (ATMs), which are geographically distributed and connected via a wide area network to a central server. Each ATM machine has a card reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader. Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN (personal identification number) matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.

Specific Requirements

1. The XYZ Bank Inc. can have many automated teller machines (ATMs), and the new software system shall provide functionality on all ATMs.
2. The system shall enable the customers of XYZ Bank Inc., who have valid ATM cards, to perform three types of transactions; 1) withdrawal of funds, 2) Query of account balance, and 3) transfer of funds from one bank account to another account in the same bank.
3. An ATM card usage shall be considered valid if it meets the following conditions:
   a) The card was issued by an authorized bank.
   b) The card is used after the start date, i.e., the date when the card was issued.
   c) The card is used before the expiration date, i.e., the date when the card expires.
   d) The card has not been reported lost or stolen by the customer, who had been issued that card.
   e) The customer provides correct personal identification number (PIN), which matches the PIN maintained by the system
4. The system shall confiscate the ATM card if it detects that a lost or stolen card has been inserted by a customer. The system shall also display an apology to the customer.
5. The system shall allow the customer to enter the correct PIN in no more three attempts. The failure to provide correct PIN in three attempts shall result in the confiscation of the ATM card
6. The system shall ask for the transaction type after satisfactory validation of the customer PIN. The customer shall be given three options: withdrawal transaction, or query transaction, or transfer transaction.
7. If a customer selects withdrawal transaction, the system shall prompt the customer to enter account number and amount to be dispensed.





Conceptual Static Model for Problem Domain: Physical Classes

OR

VI.

a) Distinguish command language and menu based user interfaces.

- Command language based interfaces

A command language-based interface as the name itself suggests, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type them in appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands

Command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them in. Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse.

Obviously, for casual and inexperienced users, command language-based interfaces are not suitable.

- Menu-based interfaces

An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu-based system. Also, if the number of choices is large, it is difficult to select from the menu. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms.

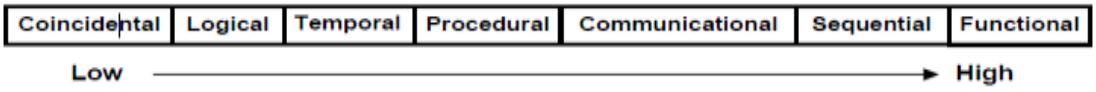
- ❖ Scrolling menu
- ❖ Walking menu
- ❖ Hierarchical menu

b) Compare cohesion and coupling.

- **Cohesion:-**

Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules.

The different classes of cohesion that a module may possess are depicted in fig.



Coincidental cohesion:-A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.

Logical cohesion:-A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc.

Temporal cohesion:-When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural cohesion:-A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

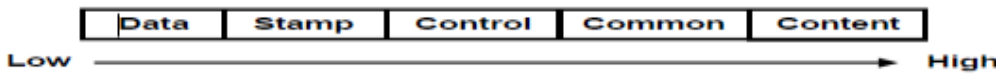
Communicational cohesion:- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

Sequential cohesion:-A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

Functional cohesion:-Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function.

- **Coupling:-**

  Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules.



  Data coupling:-Two modules are data coupled, if they communicate through a parameter.

Stamp coupling:-Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling:-Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.

Common coupling:-Two modules are common coupled, if they share data through some global data items.

Content coupling:- Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

UNIT – III

VII. Explain different types of white box approaches for testing.

White box Testing

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. A white-box testing strategy can either be coverage-based or fault- based.

One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called complementary.

- Fault-based Testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the fault model of the strategy. An example of a fault-based strategy is mutation testing.

❖ Mutation testing:-

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as

a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant.

Coverage based Testing

A coverage base testing strategy attempts to execute certain elements of a program. Popular examples of coverage based testing strategies are statement coverage, branch coverage, and path coverage based testing.

- Statement coverage:

  The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc.

- Branch coverage:

  In the branch coverage-based testing strategy, test cases are to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program's control flow graph is traversed at least once.

- Condition coverage

  In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values.

- Path coverage:

  The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

  - Control flow graph :

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program.

- Data flow-based testing:-

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.

OR

VIII. (a) Explain the debugging approaches.

- Brute Force Method:

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

- Backtracking:

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

- Cause Elimination Method:

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

- Program Slicing:

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002].

(b) Explain reliability and reliability metrics.

Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product.

• Rate of occurrence of failure (ROCOF):- ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.

• Mean Time To Failure (MTTF):- MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t1, t2, …, tn. Then, MTTF can be calculated as 1 1 (1) n ii itt n +− = − ∑ . It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements and the clock is stopped at these times.

• Mean Time To Repair (MTTR):- Once failure occurs, some timeis required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

• Mean Time Between Failure (MTBR):- MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.

UNIT – IV

IX.

a) Outline the benefits of CASE.

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

• A key benefit arising out of the use of a CASE environment is cost saving through all development phases. Different studies carry out to measure the impact of CASE put the effort reduction between 30% to 40%.

• Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.

• CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced to a great extent.

• CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.

• CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.

• Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

b) Summarize the characteristics of CASE tools.

❖ Hardware and environmental requirements

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, it can be

emphasized on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and this can be chosen for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients who run different modules access data dictionary through this server. The data dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common.

The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows to share information between users and projects. The data dictionary provides consistent view of all project entities, e.g. a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view of the data dictionary. This means that it should allow back up/restore, copy, cleaning part of the data dictionary, etc.

The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

- Documentation support

The deliverable documents should be organized graphically and should be able to incorporate text and diagrams from the central repository. This helps in producing up-to-date documentation. The CASE tool should integrate with one or more of the commercially available desktop publishing packages. It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

- Project management support

The CASE tool should support collecting, storing, and analyzing information on the software project's progress such as the estimated task

duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

- ❖ External interface

The CASE tool should allow exchange of information for reusability of design. The information which is to be exported by the CASE tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information. It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

- ❖ Reverse engineering

The CASE tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code. If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

- ❖ Data dictionary interface

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

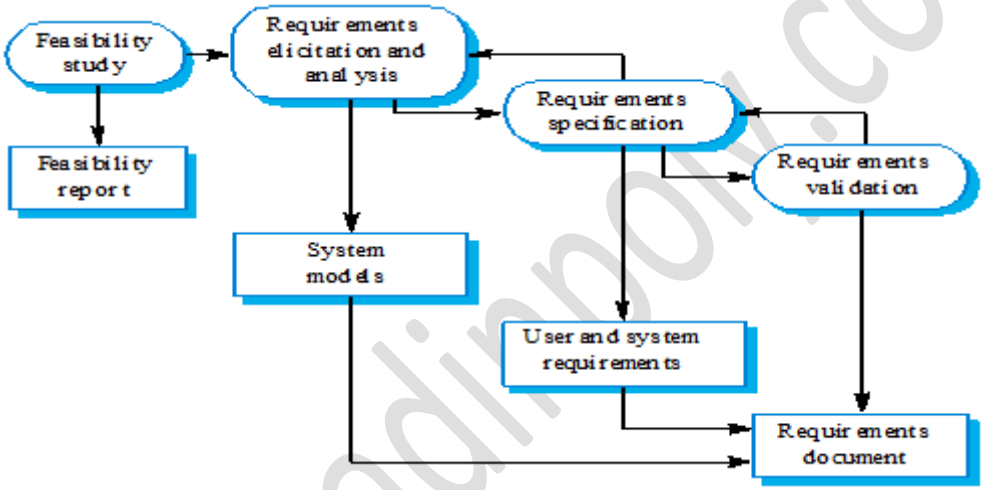
X. Illustrate a case study to build a software that automates the library system in your college. Highlights design and testing phases.

Objectives

- To describe the principal requirements engineering activities and their relationships
- To introduce techniques for requirements elicitation and analysis
- To describe requirements validation and the role of requirements reviews
- To discuss the role of requirements management in support of other requirements engineering processes

Requirements engineering processes

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of generic activities common to all processes
    - Requirements elicitation;
    - Requirements analysis;
    - Requirements validation;
    - Requirements management.



Feasibility studies

- A feasibility study decides whether or not the proposed system is worthwhile or doable.
- A short focused study that checks
    - If the system contributes to organisational objectives;
    - If the system can be engineered using current technology and within budget;
    - If the system can be integrated with other systems that are used.

Feasibility study implementation

- Based on information assessment (what is required), information collection and report writing.
- Questions for people in the organisation
    - What if the system wasn't implemented?

- What are current process problems?
- How will the proposed system help?
- What will be the integration problems?
- Is new technology needed? What skills?
- What facilities must be supported by the proposed system?

Elicitation and analysis

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

Problems of requirements analysis

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

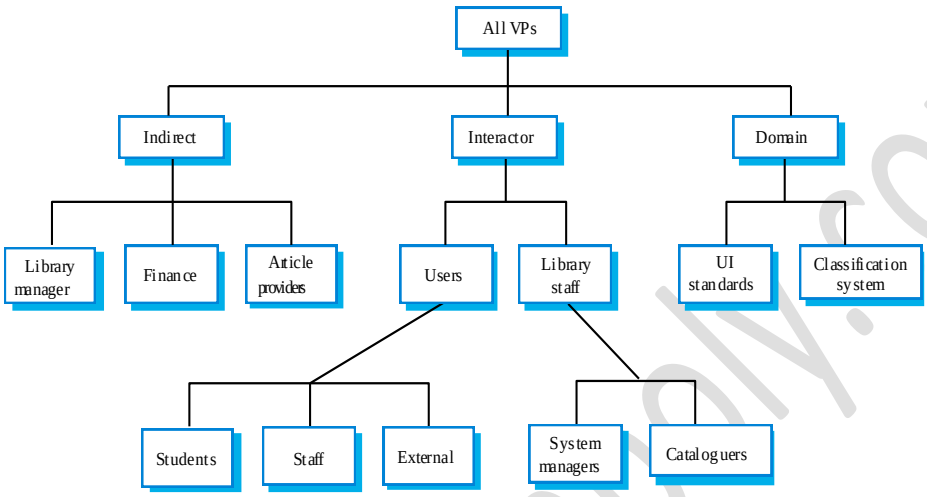
Process activities

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

Requirements discovery

- The process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems (templates).

LIBSYS viewpoint hierarchy



Interviewing

- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- There are two types of interview
  - Closed interviews where a pre-defined set of questions are answered.
  - Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.

Scenarios

- Scenarios are real-life examples of how a system can be used.
- They should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.

Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.

- Requirements error costs are high so validation is very important
    - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking

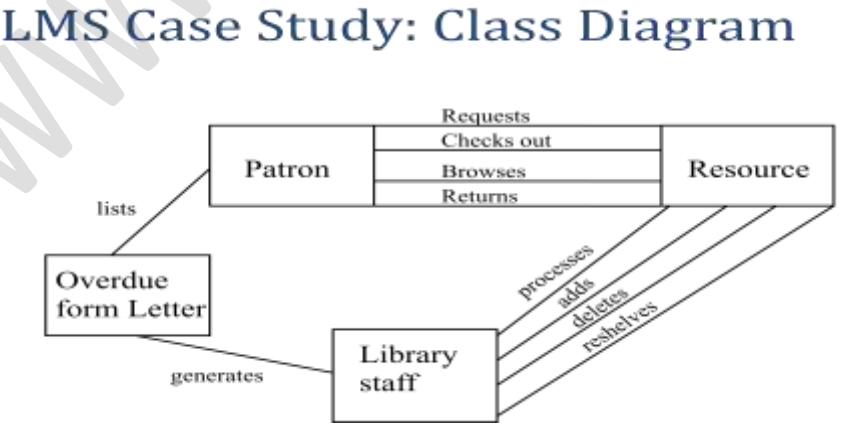
- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology
- Verifiability.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

Requirements validation techniques

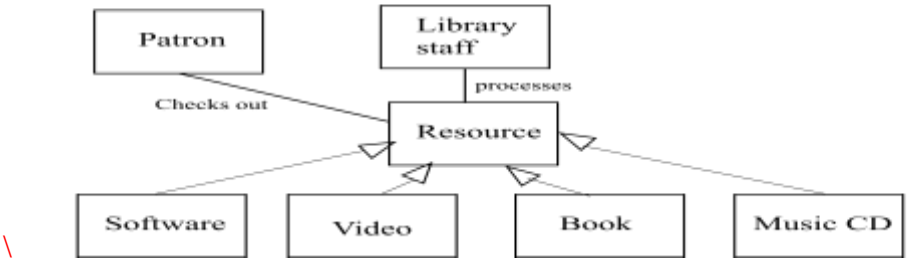
- Requirements reviews
    - Systematic manual analysis of the requirements.
- Prototyping
- Using an executable model of the system to check requirements.
- Test-case generation
- Developing tests for requirements to check testability

Requirements reviews

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



LMS Case Study: Class Diagram for Check out Resource



\

Supporting Software Tools

- § Word Processing, for Reporting
- § Spreadsheets, for Data Analysis
- § Database Management, for Data Storage
- § PowerPoint, for Data Presentation
- § Computer Aided Systems Analysis, for Methods
- § The Library Planning Model, for Evaluation
- § Project Managers, for Planning & Control

Computer Aided Analysis Methods

- § Flow charting
- § Data Flow Diagrams
- § State Transition Diagrams
- § Structure Charts
- § Entity-Relationship Diagrams (ERDs)
- § Data Dictionaries

The Library Planning Model

- § Represent Workloads in User Services
- § Represent Workloads in Technical Processing

- Estimate Staff Requirements for Workloads
- Account for Overhead, G&A, and Expenses
- Determine Requirements for Facilities
- Allocate to Means for Storage and/or Access
- Apply various Models of Library Operations
- Optimize Means for Inter-Library Cooperation
- Evaluate Future for Information Distribution
- Assess National Information Development