KVSB - Dr.K.V.SUBBA REDDY

SCHOOL OF BUSINESS MANAGEMENT

MCA & MBA COLLEGES

Institute Name:	**Dr. K. V. Subba Reddy School of Business Management**
College Code:	**JJ**
ICET Code:	**KVSB**
Name of Programmes:	**MBA & MCA**

SUBJECT: SOFTWARE ENGINEERING

SOFTWARE ENGINEERING

UNIT-I

Basic concepts in Software Engineering and Software Project Management:

Basic Concepts: abstraction Versus decomposition, evolution of Software Engineering techniques, Software development life cycle (SDLC) models: Iterative Waterfall model, Prototype model, Evolutionary model, Spiral model, Agile Models, Software Project Management: Project planning, project estimation, COCOMO, Halstead's Software science, project scheduling, staffing, Organization and team structure, risk management, configuration management.

BASIC CONCEPTS: ABSTRACTION Vs DECOMPOSITION

Abstraction is the selective examination of certain aspects of a problem while ignoring the remaining aspects of the problem. The main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress the aspects of the problem that are not relevant for the given purpose. Abstraction is a powerful mechanism for reducing complexity of software. Abstraction can be viewed as a way of increasing software productivity.

Decomposition is one of the four cornerstones of Computer Science. It involves breaking down a complex problem or system into smaller parts that are more manageable and easier to understand. The smaller parts can then be examined and solved, or designed individually, as they are simpler to work with.

EVOLUTION OF SOFTWARE ENGINEERING TECHNIQUES:

Evolution of an Art to an Engineering discipline :-

Software Engg. principles have evolved over the past fifty years with contributions from numerous researchers and s/w professionals. The early programmers used an exploratory programming style. In modern s/w industry programmers rarely make use of esoteric knowledge.

A solution to the s/w crisis :

Software Engineering appears to be among the few options available to tackle the present s/w crisis. organizations are spending larger and larger portions of their budget on s/w. s/w products are difficult to alter, debug and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. It is believed that the only satisfactory solution to the present s/w engg. practices among the engineers, coupled with further advancements in the s/w engg. discipline itself.

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC) MODELS:

A s/w life cycle is the series of identifiable stages that a s/w product undergoes during its lifetime. The first stage in the life cycle of any s/w product is usually the feasibility study stage. Commonly, the subsequent stages are: requirements analysis and specification, design, coding, testing and maintenance. Each of these stages is called a life-cycle phase.

A s/w lifecycle model is a descriptive and diagrammatic representation of the s/w lifecycle. A life-cycle model represents all the activities required to

make a s/w product transit through its life-cycle phases.

CLASSICAL WATERFALL MODEL

The Classical Waterfall Model is intuitively the most obvious way to develop S/w. We can consider this model to be a theoretical way of developing s/w. All other life-cycle models are essentially derived from the classical waterfall model.

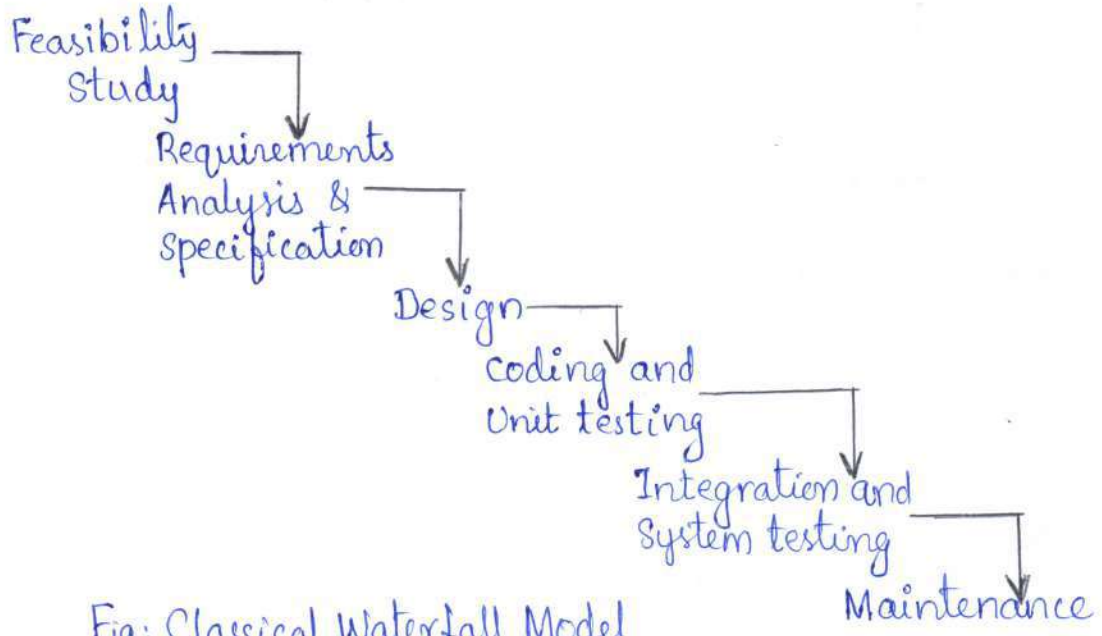


Fig: Classical Waterfall Model

The classical waterfall model divides the life-cycle into 6 phases. The name of this model is justified by its diagrammatic representation which resembles a cascade of waterfalls. The different phases of this model are: Feasibility Study, Requirements Analysis and specification, Design, Coding and unit testing, Integration and system testing and maintenance. The different phases starting from the feasibility study to the integration and system testing phase are known as development phases. The part

of the life-cycle model between the feasibility study and product testing and delivery is known as the development part. At the end of the development part of the life-cycle, the product becomes ready to be delivered to the customer. The maintenance phase commences after completion of the development phases.

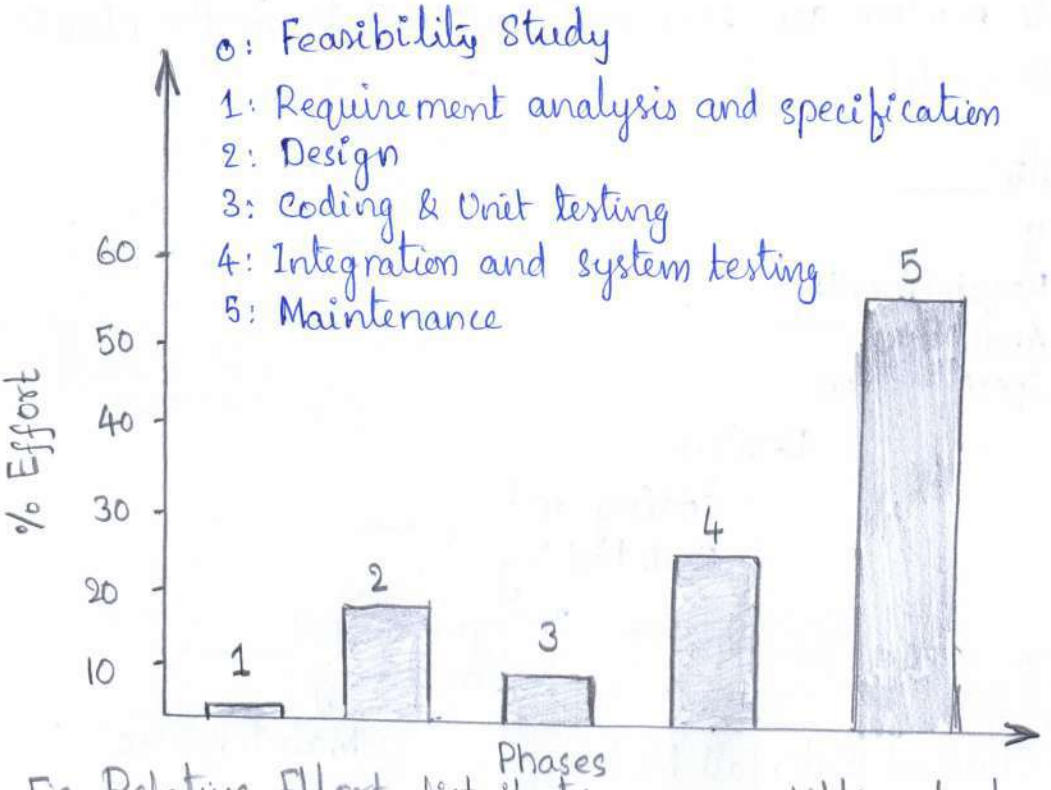


Fig. Relative Effort distribution among different phases of a typical product

1. Feasibility Study :- determines whether the s/w product financially and technically flexible to develop.
2. Requirement analysis and specification : the customer requirements identified during the requirements gathering and analysis activity are organized into a Software Requirement Specification (SRS) document.
3. Design : in this phase the s/w architecture is derived from the SRS document.
4. Coding and Unit testing : in this phase the translation of the s/w design into source code.

5. Integration and System Testing: the modules are integrated in a planned manner, and system testing is usually consists of three different kinds of testing activities: α-testing, β-testing and acceptance testing.

6. Maintenance: in this phase the activities carried are correcting errors, improving the implementation of the system and porting the s/w to work in a new environment.

ITERATIVE WATERFALL MODEL

This model is extracted from Classical Waterfall Model. An important technique used is to conduct reviews after every milestone. Inspite of best effort put in to detect in the same phase in which they were committed, some errors do escape detection and may get noticed only in the later phases.

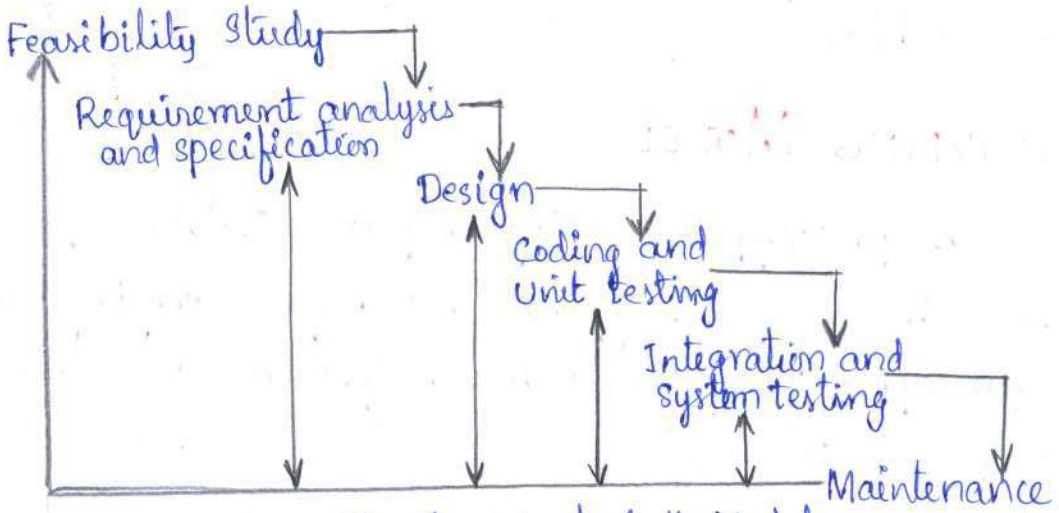


Fig: Iterative Waterfall Model.

This necessitates rework of the already completed phases. Thus, the effort spent on a phase may require some rework later due to errors in that phase which were detected in the later phases. Inspite of this, the final documents for the product should be written as if the

product was developed using a pure classical waterfall as suggested by the author Parnas.

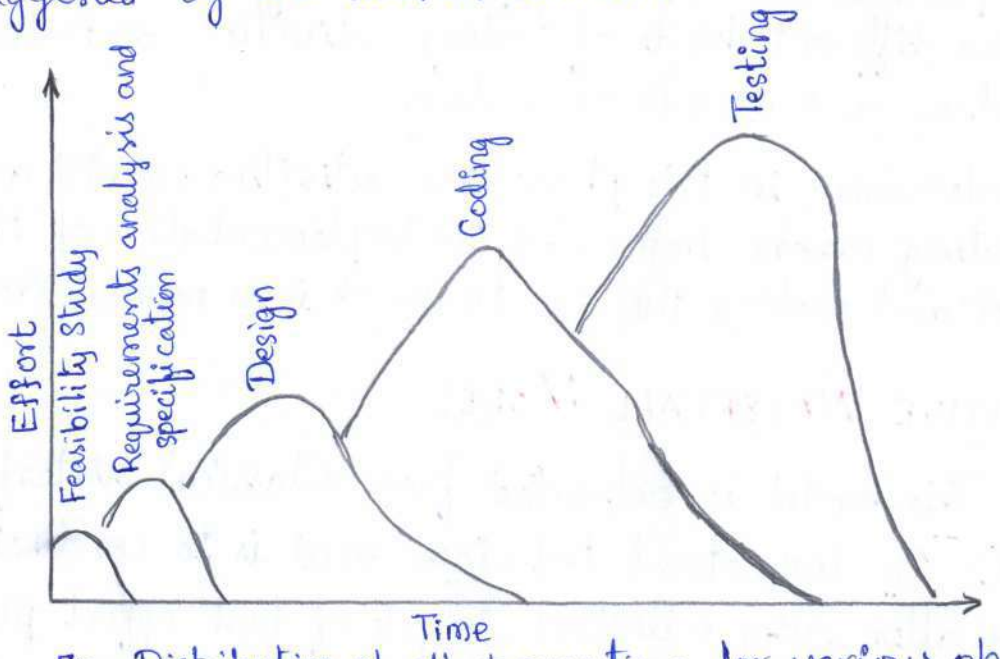


Fig: Distribution of effort over time for various phases in the iterative waterfall Model.

If the final document is written to reflect a classical waterfall model of development, comprehension of the system documents becomes easy.

PROTOTYPING MODEL

A prototype is a toy implementation of a system. This model suggests that before carrying out the development of the actual software, a working prototype of the system should be built. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compare to the actual s/w. A prototype is built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. A prototype usually turns out to be a very crude version of the actual system, also it is useful in developing the Graphical User Interface (GUI).

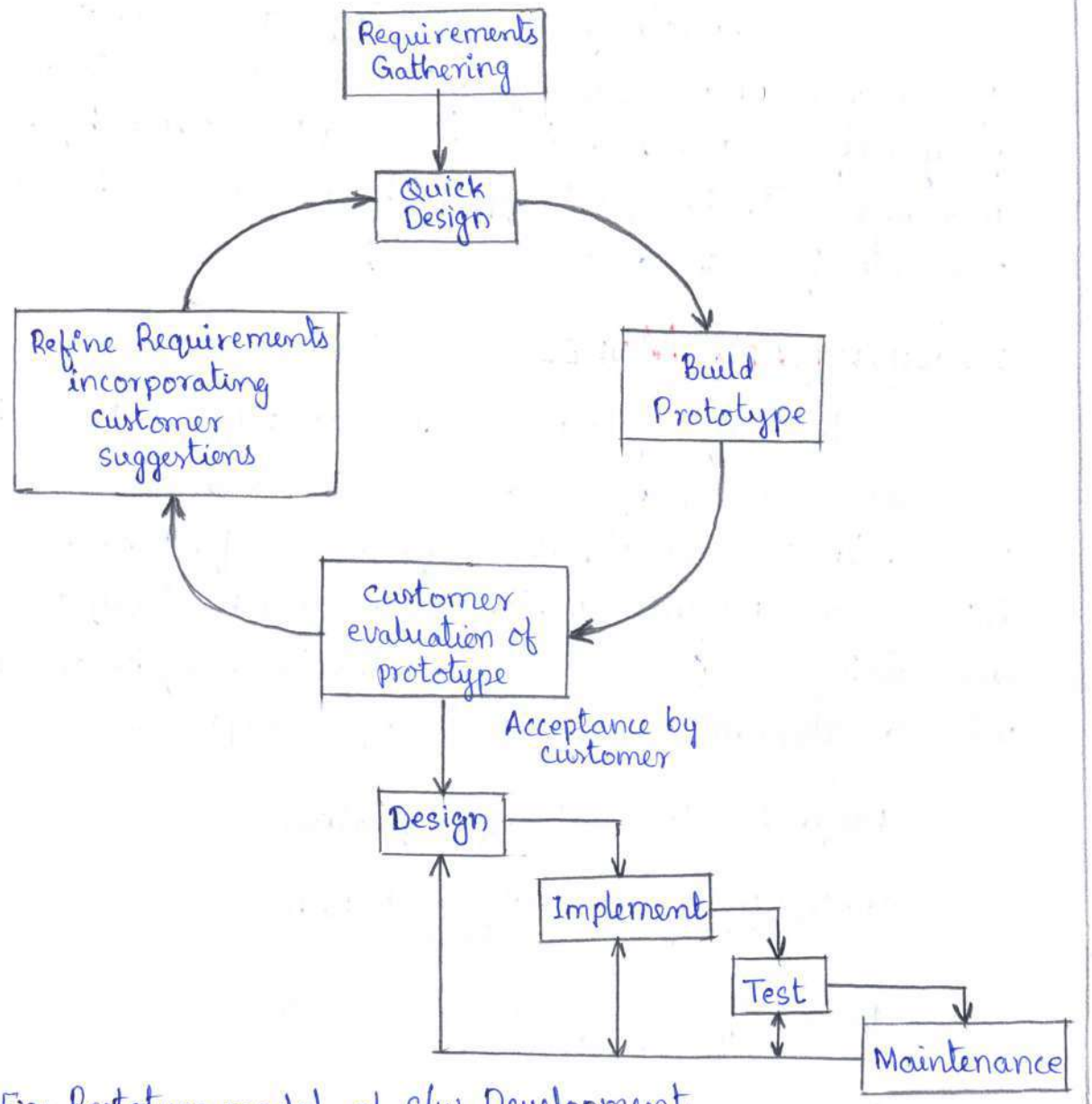


Fig: Prototype model of S/W Development

In this model, product development starts with an initial requirements gathering phase. A quick design carried out and a prototype is built. The developed prototype is submitted to the customer for his evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system.

By constructing the prototype and submitting it for user evaluation, many customer requirements get properly defined and technical issues get resolved by experimenting with the prototype. This minimizes the change requests from the customer and the associated redesign costs.

EVOLUTIONARY MODEL

This life-cycle model is also referred to as the successive versions model and sometimes as the incremental model. In this model, the software is first broken down into several modules which can be incrementally constructed and delivered. Each evolutionary version may be developed using an iterative waterfall model of development.

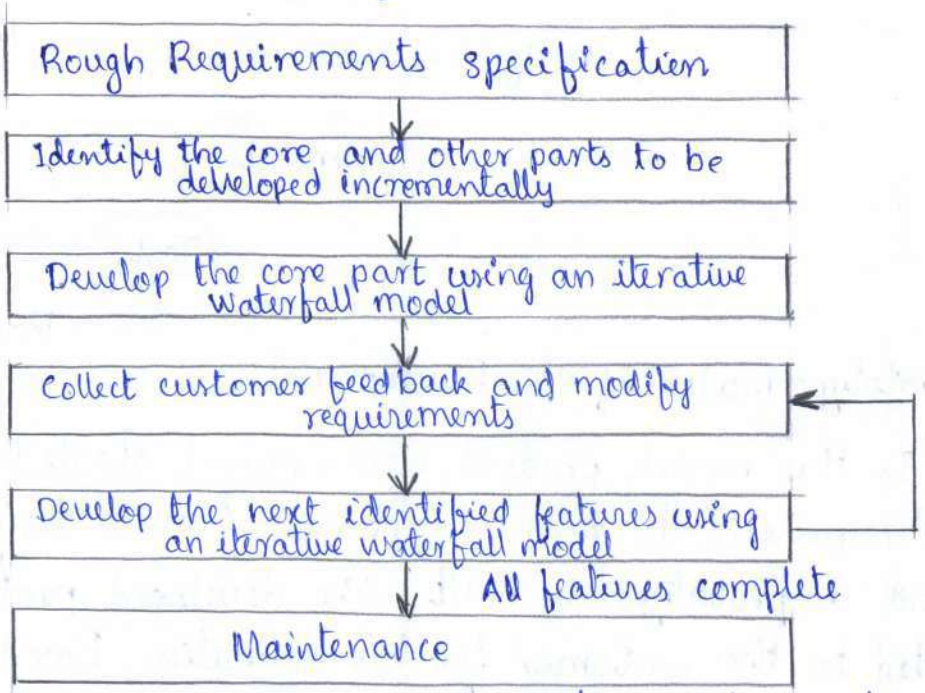


Fig: Evolutionary model of s/w Development.

It is difficult to divide the problem into several functional units which can be incrementally implemented and delivered. Therefore, this model normally useful for only large size products. This model is very popular for object-oriented

development projects, because the system can easily be partitioned into standalone units in terms of the object.

SPIRAL MODEL

The Spiral model of s/w development 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.

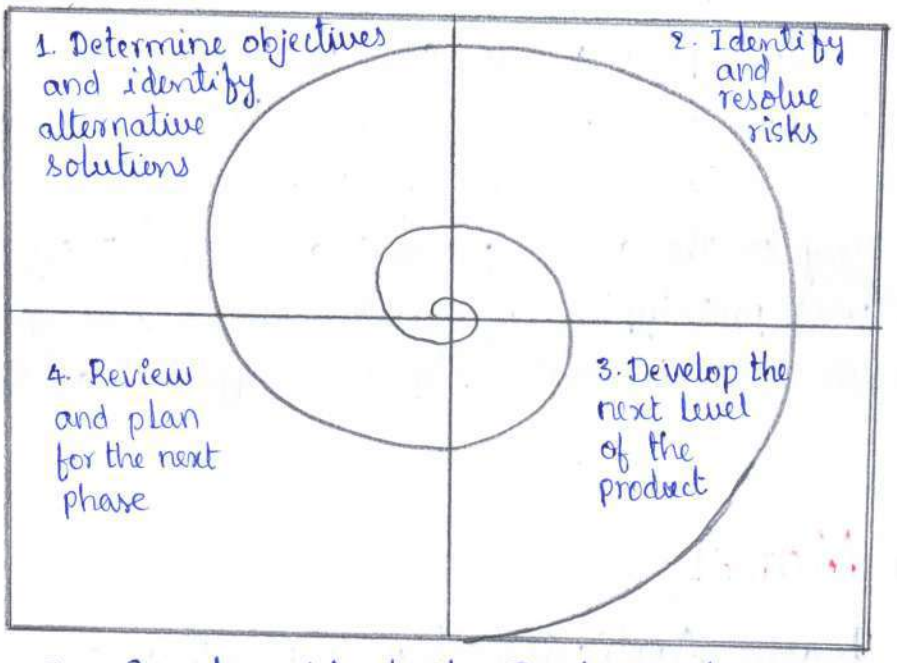


Fig: Spiral model of S/w Development

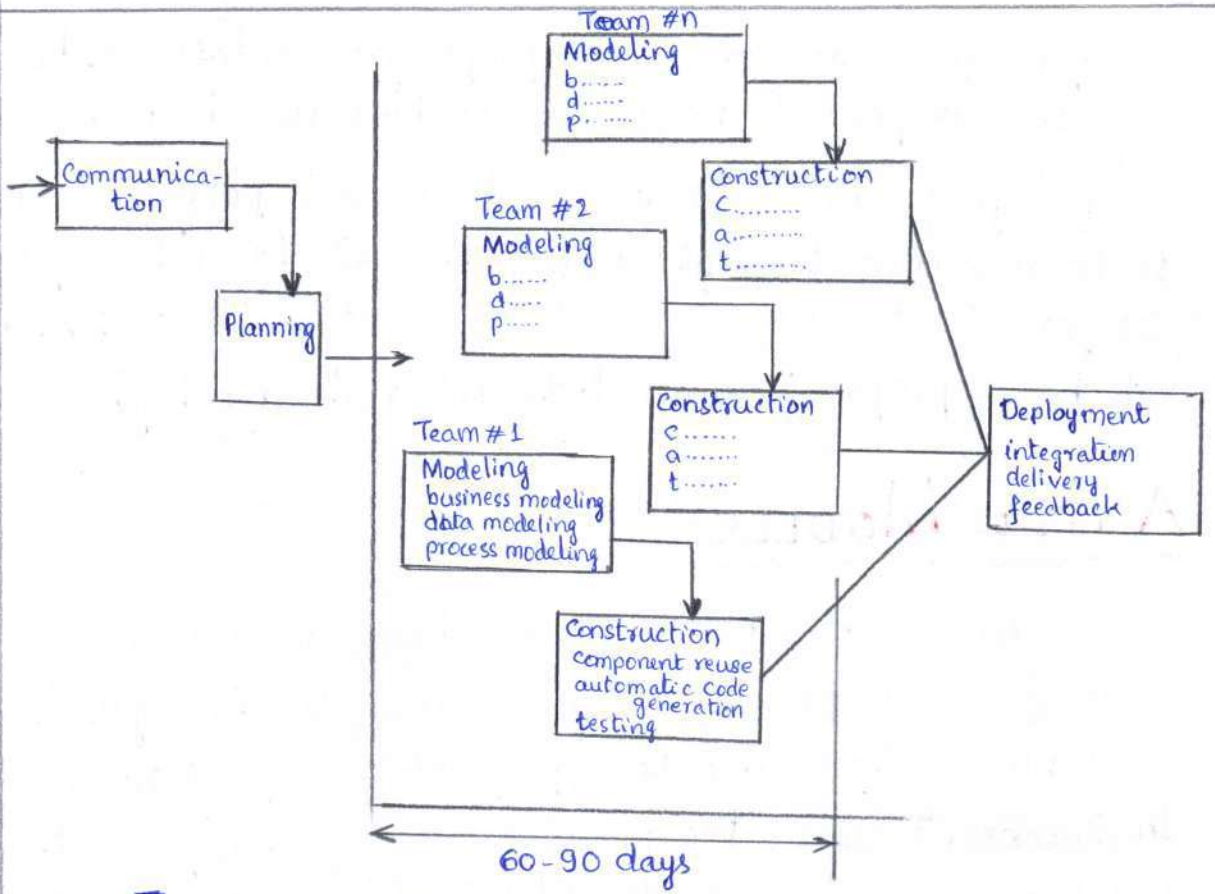
For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design and so on. This model is much more flexible compared to the other models. This phase split into four quadrants or sectors, the first sector identifies the objectives of the phase and the alternative solutions possible for the phase under consideration. During the second quadrant, the alternative solutions are evaluated to select the best solution possible. For the chosen solution, the potential risks are identified and dealt with by developing an appropriate prototype.

The risk can be resolved by building a prototype of the data access subsystem. Thus, the spiral model provides direct support for coping with the project risks. The third quadrant consist of developing and verifying the next level of the project risks. Activities during the fourth quadrant concern reviewing the result of the stages traversed so far with the customer and planning the next iteration around the spiral.

The Spiral model can be viewed as a meta model, since it subsumes all the models. The spiral model uses a prototyping approach by first building a prototype before embarking on the actual product development effort. The Spiral model uses prototyping as a risk reduction mechanism and also retains the systematic step-wise approach of the waterfall model.

RAD MODEL

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



The RAD approach maps into the generic framework activities presented earlier. Communication works to understand the business problem. Planning is essential because multiple S/W teams work in parallel on different system functions. Modeling encompasses three major phases - business modeling, data modeling and process modeling. Construction model emphasizes the use of preexisting S/W components and the application of automatic code generation.. Finally the deployment model establishes a basis for subsequent iterations, if required.

Like all process models, RAD approach has drawbacks : (1) For large, but scalable projects, RAD requires sufficient human resources to create right number of RAD teams. (2) If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail.

(3) If a system is cannot be properly modularized, building the components necessary for RAD will be problematic. (4) If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work. (5) RAD may not be appropriate when technical risks are high.

AGILE MODELS

Agility has become today's buzz word when describing a modern S/W process. Every one is agile. An agile team is a nimble team able to appropriately respond to changes. In Jacobson's view, the pervasiveness of change is the primary driver for agility. S/W Engineers must quick on their feet if they are to accommodate the rapid changes.

1. Extreme Programming (XP) :

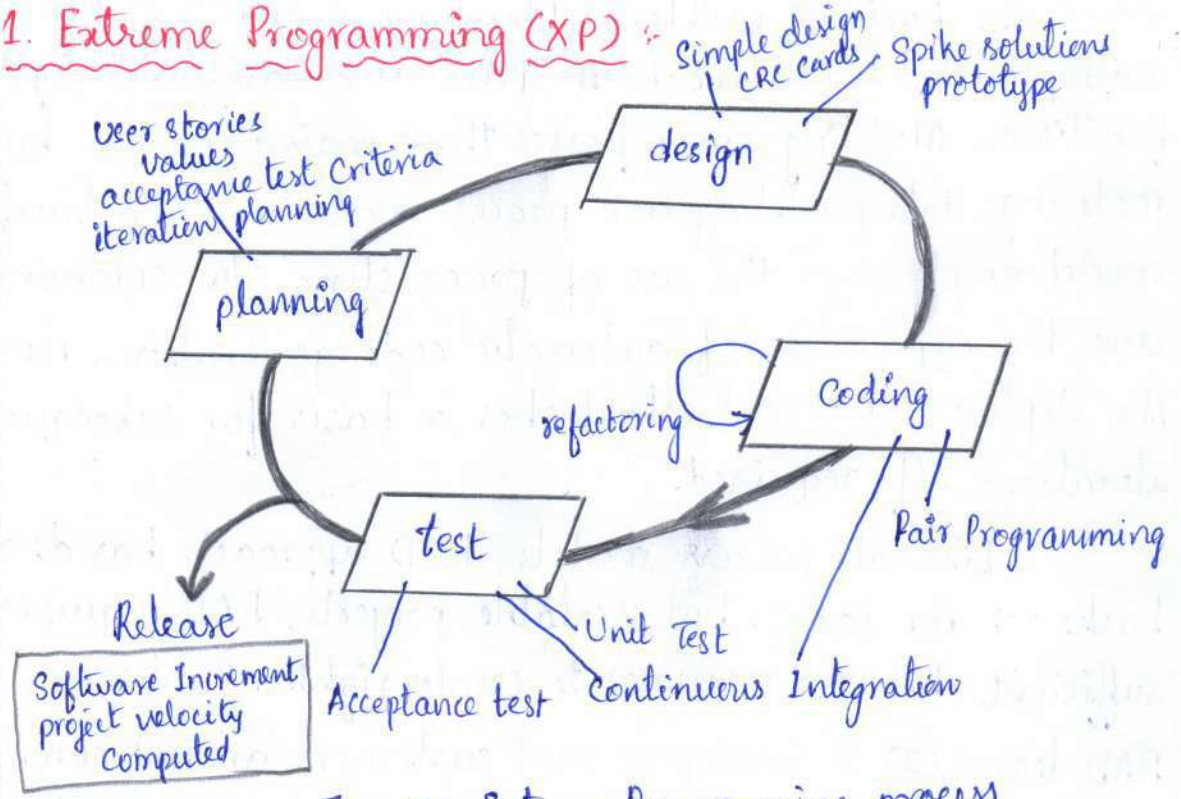


Fig: The Extreme Programming process

Extreme Programming (XP) is the most widely used agile process. Originated as four framework activities - planning, design, coding and testing - XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent s/w releases delivering features and functionalities that have been described and prioritized by the customer.

2. Adaptive Software Development :

This model stresses human collaboration and team self-organization. Organized as three framework activities - speculation, collaboration and learning. ASD uses an iterative process that incorporates adaptive cycle planning, relatively rigorous requirement gathering methods, and an iterative development cycle that incorporates customer focus groups and formal technical reviews as real-time feedback mechanism.

3. Scrum :

Scrum emphasizes the use of a set of s/w process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the scrum team to construct a process that is adapted to the needs of the project.

4. Crystal :

Crystal is a family of agile process models that can be adopted to the specific characteristics of a project. Like other agile approaches, crystal adopts an iterative strategy but adjusts the rigor of the objects process to accommodate projects of different sizes and complexities.

5. Feature Driven Development (FDD):

FDD is somewhat more formal than other agile methods, but still maintains agility by focusing the project team on the project development of features - client-valued functions that can be implemented in two weeks or less. FDD provides greater emphasis on project and quality management than other agile approaches. Agile modeling suggests that modeling is essential for all systems but that the complexity, type and size of the model must be turned to the s/w to be built. By proposing a set of core and supplementary modeling principles, Agile modeling provides useful guidance for the practitioner during analysis and design tasks.

SOFTWARE PROJECT MANAGEMENT

SPM is crucial to the success of any s/w project. In the past, several s/w projects have failed not for want of competent technical professionals or resources but because of use of faulty s/w project management practices. Therefore, it is important to carefully learn the latest s/w project management techniques.

The main goal of SPM is to enable a group of s/w engineers to work efficiently towards successful completion of project. The responsibities of s/w project manager are project proposal writing, cost estimation, scheduling, staffing, process tailoring, monitoring and control, configuration management, risk management, interfacing with clients, managerial report writing and presentations. The manager take the overall responsibility of steering a project to success and team building are largely acquired through experience.

PROJECT PLANNING

Once a project is found to be feasible, s/w 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:

- Estimating some basic attributes of the project
  - Cost: How much will it cost to develop the project?
  - Duration: How long will it take to complete the development?
  - Effort: How much effort would be required?

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

- Scheduling manpower and other resources
- Staff organization and staffing plans
- Risk identification, analysis and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

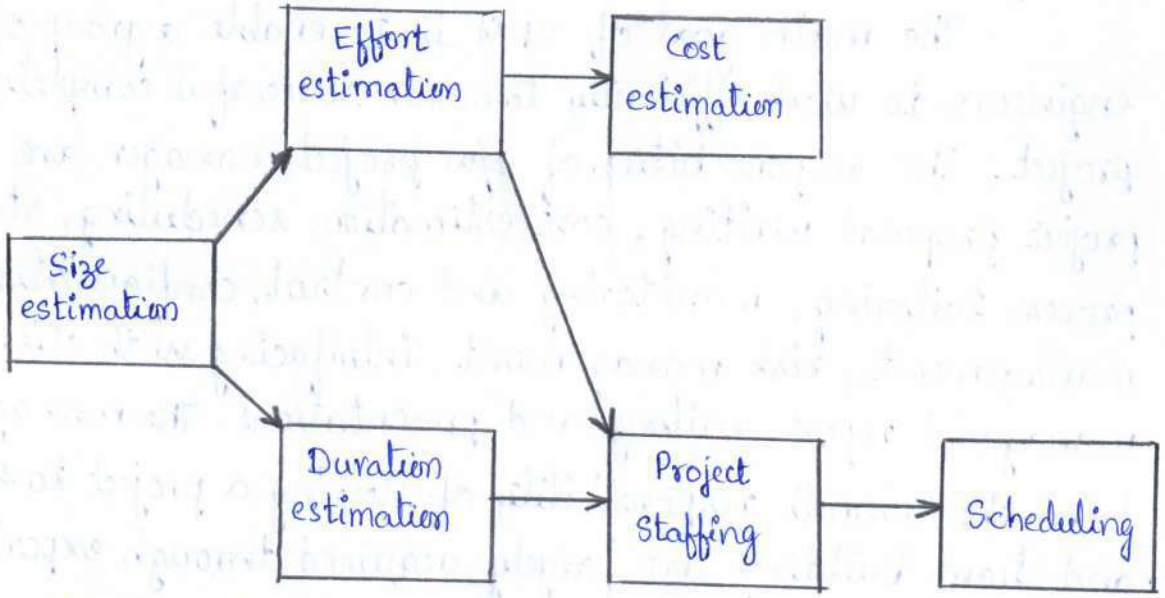


Fig: Precedence ordering among planning activities

Estimation of effort, cost, resource, and project duration are also very important components of project planning. Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates results in schedule slippage. 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.

The SPMP Document

Once project planning is complete, project managers document their plans in a S/w Project Management Plan (SPMP)

document. The following are the different items that the SPMP document should discuss. This list can be used to organize the SPMP document.

Organization of the 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 & 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, Istallation and Maintenance Plan.

PROJECT ESTIMATION

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a S/W project. In order to be able to accurately estimate the project size, we need to define some appropriate metric or unit in terms of which we can express the project size. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

1. Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. This metric is very popular, being the simplest to use. Using LOC, the project size is estimated by counting the number of source instructions in the developed program. 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 submodules and so on. However LOC as a measure of problem size has several shortcomings:

1. LOC gives a numerical value of problem size that can vary with individual coding style.
2. A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it.
3. LOC measure correlates poorly with the quality and efficiency of the code.
4. LOC metric penalizes use of higher-level programming languages, code reuse, etc.
5. LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities.
6. It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed.

2. Function Point Metric

Function Point metric was proposed by Albrecht in 1983. This metric overcomes many of the LOC metric. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a s/w product directly from the problem specification

The conceptual idea underlying the Function Point metric is that the size of a s/w product is directly dependent on the number of different functions or features it supports. Albrecht postulated that in addition to the number of basic functions that a s/w performs, the size is also dependent on the number of files and the number of interfaces.

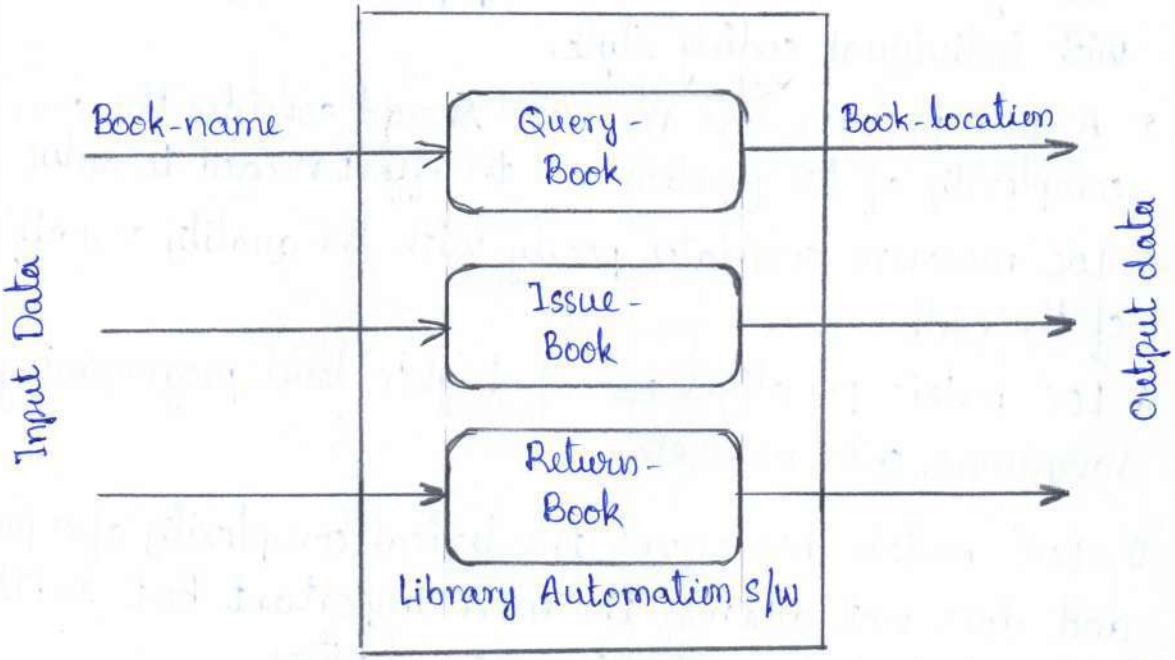


Fig: System function as a map of input data to output data

Function Point is computed in two steps. The first step is to compute the unadjusted function point (UFP) or Universal function point.

UFP = (No. of inputs) * 4 + (No. of outputs) * 5 + (No. of inquiries) * 4 + (No. of files) * 10 + (No. of interfaces) * 10

- No. of Inputs :- Each data item input by the user is counted. Data input should be distinguished from user inquiries.
- No. of outputs :- The outputs considered refer to reports printed, screen outputs, error messages produced, etc.
- No. of inquiries :- Number of inquiries is the number of distinct interactive queries which can be made by the user.
- No. of files :- A logical file means groups of logically related data.
- No. of interfaces :- The interfaces considered are the interfaces used to exchange information with other systems.

Feature Point Metric :-

Feature point metric is an extension to Function point metric. FPM incorporates an extra parameter into algorithm complexity. These metrics are language-independent and can be easily computed from the Software Requirement Specification (SRS) document during project planning.

COCOMO :-

Constructive Cost Estimation Model was proposed by Barry Boehm [1981]. Boehm postulated that any s/w development project can be classified into one of the following three characteristics is based on the development complexity: organic, semidetached, and embedded.

Boehm's definitions of organic, semidetached and embedded systems are elaborated below.

Organic :- Projects deal with developing a well-understood application program, the size of team is small and members are experienced.

Semidetached :- the team consists of a mixture of experienced and inexperienced staff.

Embedded :- the s/w being developed is strongly coupled to complex h/w.

Boehm provides different sets of expressions to predict the effort and development time from the size estimation given in KLOC (Kilo Lines of Source Code). One person-month (PM) is the effort an individual can typically put in a month.

According to Boehm, s/w cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

1. 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 :

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \quad \text{PM}$$
$$T_{dev} = b_1 \times (\text{Effort})^{b_2} \quad \text{months}$$

Where
KLOC is the estimated size of the s/w product expressed
a_1, a_2, b_1, b_2 are constants for each category of s/w products
Tdev is the estimated time to develop the s/w, expressed in months
Effort is the total effort required to develop the s/w product, expressed in person-months (PM).

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, it is considered to be nLOC.

Estimation of development effort :-

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

Organic : $\text{Effort} = 2.4 (\text{KLOC})^{1.05}$ PM
Semi-detached : $\text{Effort} = 3.0 (\text{KLOC})^{1.12}$ PM
Embedded : $\text{Effort} = 3.6 (\text{KLOC})^{1.20}$ PM

Estimation of development time :-

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

Organic : $T_{dev} = 2.5 (\text{Effort})^{0.38}$ months
Semi-detached : $T_{dev} = 2.5 (\text{Effort})^{0.35}$ months
Embedded : $T_{dev} = 2.5 (\text{Effort})^{0.32}$ months

It is important to note that the effort and duration estimations obtained using the COCOMO model are called the normal effort estimate and the normal duration estimate.

Example :- Assume that the size of an organic type S/w product has been estimated to be 32,000 lines of source code. Assume that the salary (average) of S/w engineers is Rs. 15,000 per month. Determine the effort required to develop the S/w product and the nominal development time.

Sol :- From the basic COCOMO estimation formula for organic S/w:

Effort $= 2.4 \times (32)^{1.05} = 91$ PM

Nominal development time $= 2.5 \times (91)^{0.38} = 14$ months

Cost required to develop the product $= 14 \times 15{,}000 = Rs.\ 210{,}000$.

2. Intermediate COCOMO :-

The Basic COCOMO model assumes that effort and development time are functions of the product size alone. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained through the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of S/w development. Boehm requires the project manager to rate the 15 different parameters for a particular project on a scale of one to three. Then, depending one these ratings, suggests appropriate cost driver values, i.e. multipliers of the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items :

Product :- The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer :- the execution speed required, storage space required, etc.

Personnel :- the experience level of personnel, programming capability, analysis capability, etc.

Development Environment :- It captures the development facilities available to the developers.

3. Complete COCOMO :-

The complete COCOMO model considers the differences in the characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margins of error in the final estimate.

Let us consider the MIS development project as an example application of the complete COCOMO model. A distributed Management Information System 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

HALSTEAD'S SOFTWARE SCIENCE - AN ANALYTICAL TECHNIQUE

Halstead's Software Science is an analytical technique to measure size, development effort, and development cost of s/w products. Halstead used a few primitive program parameters to develop the expressions for the overall program length, potential minimum volume, actual volume, language level, effort and development time.

- η_1 be the no. of unique operators used in the program.
- η_2 be the no. of unique operands used in the program.
- N_1 be the total no. of operators used in the program.
- N_2 be the total no. of operands used in the program.

A label is considered to be an operator, if it is used as the target of a GOTO statement. The constructs if... then... else.. endif and a while... do are treated as single operators. A sequence operator ';' is treated as single operator

Operators and Operands for the ANSI C Language :

The following is a suggested list of operators for the ANSI C Language :

([. , -> * + ! ++ -- / % << >> < > <= >= != == & ^ | && || = *= /= += -= <<= >>= : ; ? (CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO BREAK CONTINUE RETURN etc.,

Operands are those variables and constants which are used with operators in expressions. The function name in a function definition is not counted as an operator.

Length and Vocabulary

The length of the program quantifies the total usage of all operators and operands in the program. Thus, length $N = N_1 + N_2$. Halstead's definition of the length of the program as the total number of operators roughly agrees with the intuitive notion of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, program vocabulary $\eta = \eta_1 + \eta_2$

Program Volume

The length of a program depends on the choice of the operators and operands used. Thus, while expressing program

size, the programming language used must be taken into consideration. Let us try to understand the important idea underlying the following expression

$$V = N \log_2 \eta$$

Potential Minimum Volume

The potential minimum volume V^* is defined as the volume of the most succint program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction. Thus, if an algorithm operates on input and output data $d_1, d_2, \ldots d_n$, the most succint program would be $f(d_1, d_2, \ldots d_n)$; for which $\eta_1 = 2$, $\eta_2 = \eta$. Therefore $V^* = (2 + \eta_2) \log_2 (2 + \eta_2)$.

Effort and Time

The effort required to develop a program can be obtained by dividing the program volume by the level of the programming language used to develop the code. Thus, effort $E = V/L$, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort $E = V^2/V^*$ (since $L = V^*/V$)

The programmer's time, $T = E/S$, where S is the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

Length Estimation

Halstead suggests a way to determine the length of a program utilizing the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity.

Example Prg:

Let us consider the following C program.

```
main()
{
    int a, b, c, avg ;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a+b+c)/3 ;
    printf("avg = %d", avg ) ;
}
```

The unique operators are:

main(), {}, int, scanf, &, "", "", ;, =, +, /, printf

The unique operands are:

a, b, c, &a, &b, &c, a+b+c, avg, 3, %d %d %d, avg, =, %d

Therefore, $n_1 = 12$, $n_2 = 11$

Estimated Length $= (12 * \log 12 + 11 * \log 11)$

$= (12 * 3.58 + 11 * 3.45)$

$= 43 + 38$

$= 81$

Volume = Length * log(23)

$= 81 * 4.52 = 366.$

PROJECT SCHEDULING

Scheduling the project tasks is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a S/w project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time duration necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path.

WORK BREAKDOWN STRUCTURE

WBS is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks needed to be carried out in order to solve a problem. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities require approximately two weeks to develop.

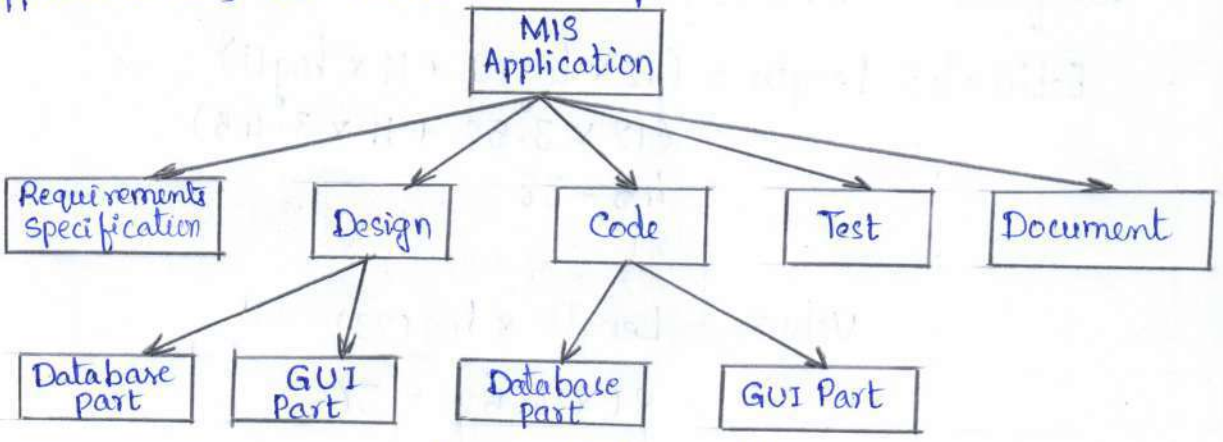


Fig. WBS structure of MIS problem.

While breaking down a large task into smaller subtasks, the manager has to make some hard decisions. If a task is broken down into a large number of very small activities, these can be distributed to a larger number of engineers. However, it is not useful to subdivide tasks into units which take less than a week or two to execute.

ACTIVITY NETWORKS AND CRITICAL PATH METHOD

An activity network shows the different activities making up a project, their estimated durations, and inter-dependencies. Each activity is represented by a rectangular node and the duration of the activity is shown in the below diagram.

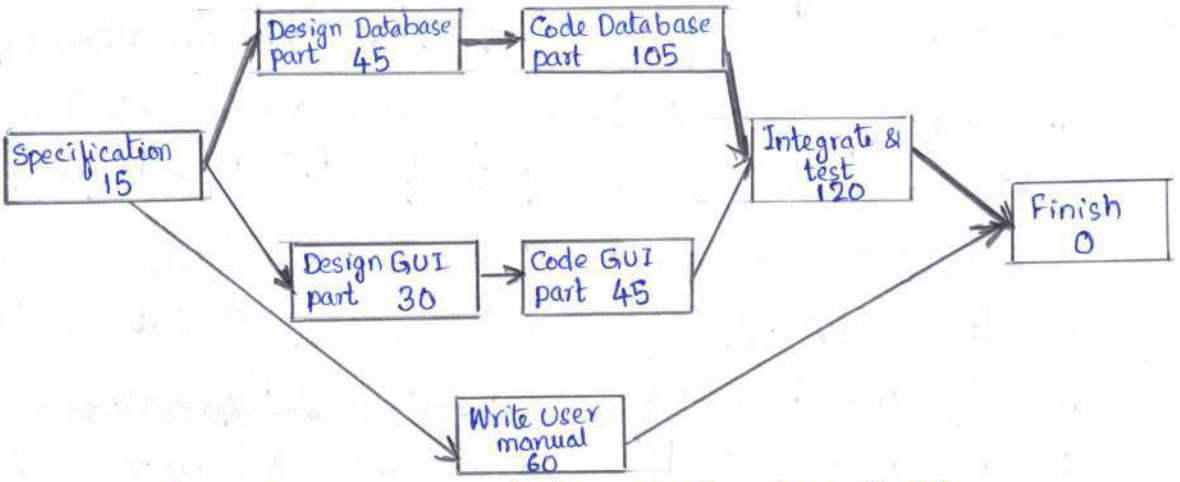


Fig: Activity Network representation of the MIS Problem

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign duration to different tasks. A possible alternative is to let each engineer himself estimate the time for an activity that is assigned to him. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the engineers to do a better and faster job. A good way to achieve accuracy in estimation of the task durations without

creating undue schedule pressures is to have people set their own schedules.

CRITICAL PATH METHOD (CPM)

From the activity network representation, the following analysis can be made:

- The minimum time (MT) to complete the project is the maximum of all paths from start to finish.
- The earliest start (ES) time of a task is the maximum of all paths from the start to this task.
- The latest start (LS) time is the difference between MT and the maximum of all paths from the task to the finish.
- The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task.
- The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.
- The slack time (ST) is LS - EF and equivalently can be written as LF - EF. The slack time (or float time) is the total time for which a task may be delayed before it would affect the finish time of the project. The slack time indicates the flexibility in starting and completion of tasks.
- A critical task is one with a zero slack time.
- A path from the start node to the finish node containing only critical tasks is called a critical path.

GANTT CHARTS

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware and software. Gantt charts are useful for resource planning. A Gantt chart is a special type of bar chart

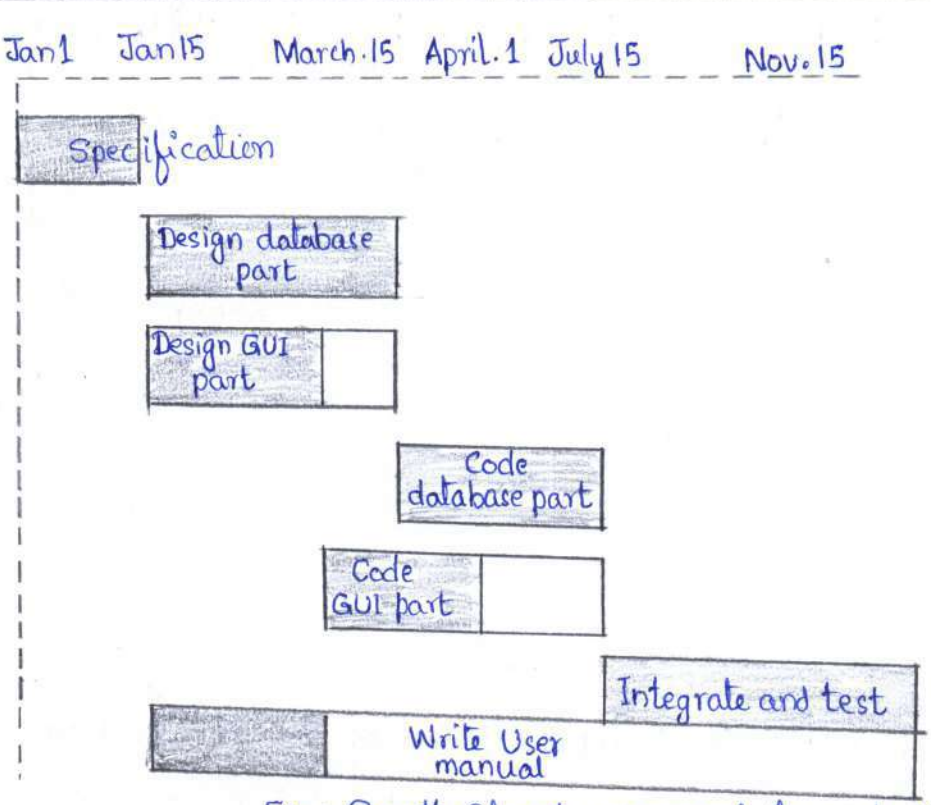


Fig: Gantt Chart representation of MIS problem

Gantt Charts used in S/W Project Management are actually an enhanced version of the standard Gantt Charts. This chart is named after its developer Henry Gantt. In the Gantt Charts used for S/W project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished.

PERT CHARTS

PERT (Project Evaluation Review Technique) charts consist of a network of boxes and arrows. The boxes represents activities and the arrows represent task dependencies PERT chart represents the statistical variations in the project estimates assuming a normal distribution. A critical

path in a PERT chart is shown by using thicker arrows

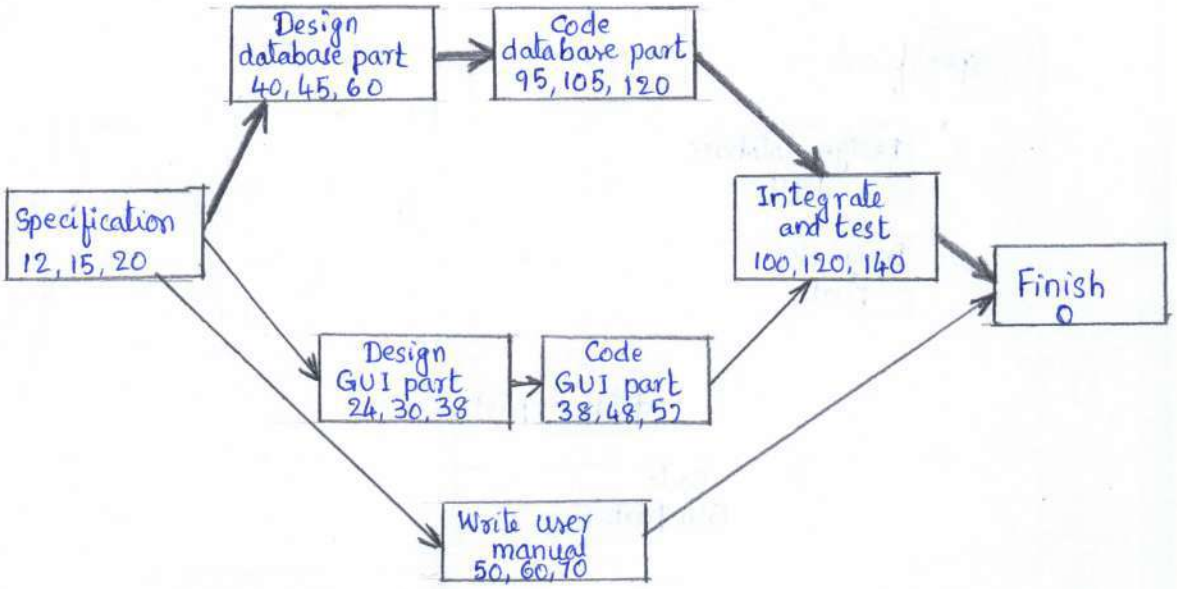


Fig: PERT chart representation of the MIS problem

PERT charts are more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since the actual durations might vary from the estimated duration, the utility of the activity diagram is limited.

PERT is useful for monitoring the timely progress of activities. Also, it is easy to identify parallel activities in a project using a PERT chart.

PROJECT SCHEDULING

Project Scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to

STAFFING

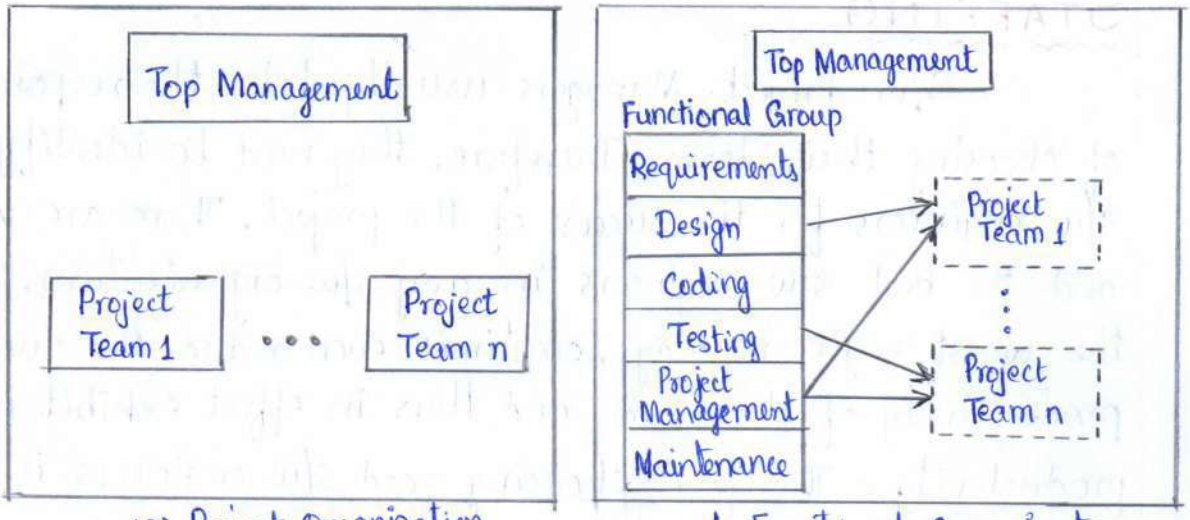
S/w Project Managers usually take the responsibility of choosing their team. Therefore, they need to identify good s/w engineers for the success of the project. There are worst and the best s/w engineers in any s/w organizations. In fact, the worst engineers may sometimes even reduce the overall productivity of the team, and thus in effect exhibit negative productivity. Therefore, choosing good s/w engineers is crucial to the success of a project.

Good s/w Engineer's attributes:

- Exposure to systematic techniques, i.e. familiarity with s/w engg. principles.
- Good technical knowledge of the project areas (Domain Knowledge)
- Good programming abilities
- Good oral, written, and interpersonal skills.
- High motivation
- Sound knowledge of fundamentals of computer science
- Intelligence
- Ability to work in a team
- Discipline, and so forth.

ORGANIZATION AND TEAM STRUCTURES

S/w organizations assign different teams of engineers to handle different s/w projects. There are essentially two broad ways in which a s/w development organization can be structured: functional format and project format. In the project format, the development staff are divided based on the project for which they work.



(a) Project Organization (b) Functional Organization

In the functional format, different teams of programmers perform different phases of a project. The functional format requires considerable communication among the different teams, because the work of one team must be clearly understood by the subsequent teams working on the project.

In the project format, a set of engineers are assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities.

TEAM STRUCTURE

Team structure addresses the issue of organization of the individual project teams. There are three formal team structures: democratic, chief programmer, and the mixed team organization.

Democratic Team: The democratic team does not enforce any formal team hierarchy. Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership. A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects.

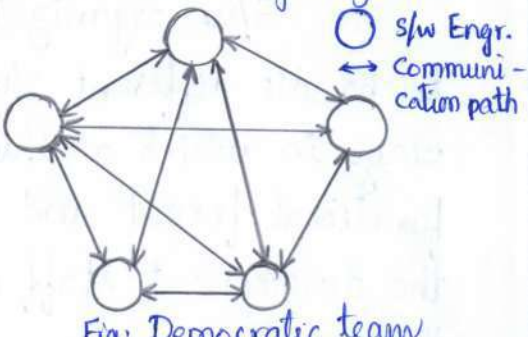
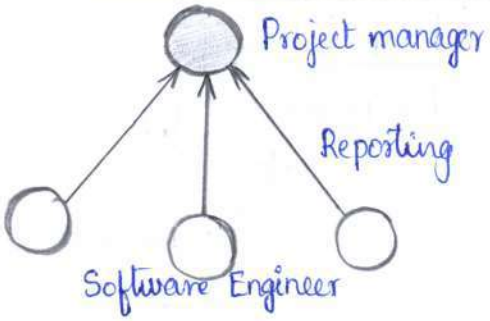


Fig: Democratic team

Chief Programmer Team: In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. The chief programmer provides an authority, and this structure is arguably more efficient than that of the democratic team for well-understood problems.



Mixed Control Team Organization:

The mixed team organization, draws upon the ideas from both the democratic organization and the chief programmer organization. This team organization incorporates both hierarchical reporting and democratic setup. The mixed control team organization is suitable for large team sizes. This team structure is extremely popular and is being used in many s/w development companies.

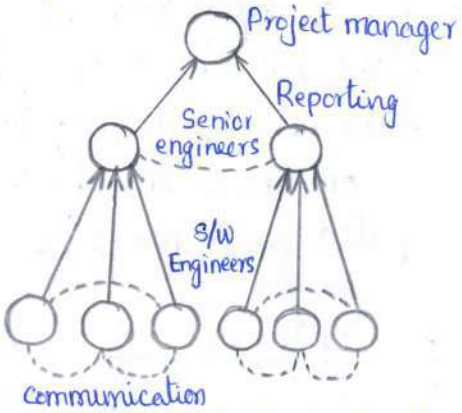


Fig. Mixed team structure

RISK MANAGEMENT

A Risk is any anticipated unfavourable event or circumstance that can occur while a project is underway. If a risk becomes true, it can hamper the successful and timely completion of a project. Therefore, it is necessary to anticipate and identify different risks that a project may be susceptible to, so that contingency plan can be prepared to contain the effect of each risk. Risk management aims reducing impact of all kinds of risks. Risk management consist of three essential activities: risk identification, risk assessment, and risk containment.

Risk Identification: The impact of the risks can be minimized by making effective risk management plans. There are three main categories of risks which can affect a s/w project.

- Project risks
- Technical risks
- Business risks.

Risk Assessment: The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, each risk should first be rated in two ways:

- the likelihood of a risk coming true (r)
- the consequence of the problems associated with that risk (s)

Based on these two factors, the priority of each risk can be computed as $p = r * s$ (p = priority)

Risk Containment: After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. Different risks require different containment procedures. In fact, most risks require ingenuity on the part of the project manager in tackling them; there are three main strategies used for risk containment.

- Avoid the risk
- Transfer the risk
- Risk reduction

$$\text{Risk leverage} = \frac{\text{risk exposure before reduction} - \text{risk exposure after reduction}}{\text{Cost of reduction}}$$

CONFIGURATION MANAGEMENT

Configuration management is to control the access to the different deliverable objects. The following are some of the important problems that appear if configuration management is not used.

- Inconsistency problem when the objects are replicated.
- Problems associated with concurrent access.
- Providing a stable development environment
- System accounting and maintaining status information
- Handling variants.

Configuration Management Activities :

A project manager performs the configuration management activity by using an automated configuration management tool. This tool provides automated support for overcoming all the problems. Also enables the engineers to change the various components in a controlled manner. Configuration management is carried out through two principal activities.

- Configuration Identification : involves deciding which parts of the system should be kept track of.
- Configuration Control : ensures that changes to a system happen smoothly.

— * END OF UNIT-I * —

SOFTWARE ENGINEERING

UNIT-II

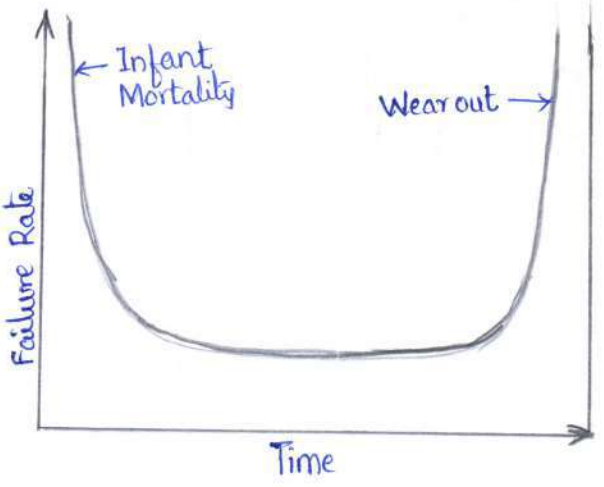
Requirements Analysis and Specification: The nature of S/W, The unique nature of Webapps, S/W Myths, Requirement gathering and analysis, S/W requirements Specification, Traceability, Characteristics of a Good SRS Document, IEEE 830 guidelines, Representing Complex Requirements using decision tables and decision trees, Overview of formal system development techniques, axiomatic specification, algebraic specification.

THE NATURE OF SOFTWARE

- Software is intangible
  - Does not exist in physical form
  - Hard to see the effort involved in its development process.
- Software is easy to reproduce
  - only costs in its development
  - Generating another copy of same S/W is easy once it is developed.
  - Building a similar S/W require less time.
- Software does not wear out
  - Deteriorates with maintenance as changes are introduced.
  - Slows down in performance as time passes.
  - Suffers when not updated regularly with respect to the surrounding environment.

Software Characteristics:

1. S/W is developed or engineered, it is not manufactured in the classical sense.
2. S/W doesn't wear out.
   The bath tub curve indicates that h/w exhibits relatively high failure rates early in its life: defects are corrected and the failure rate drops. S/W doesn't wear out but its does deteriorate.



3. Although the industry is moving toward component-based assembly, most s/w continues to be custom built.

THE UNIQUE NATURE OF WEBAPPS

In the early days of the World Wide Web (1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.

Today, webapps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications due to the development of HTML (Hyper Text Markup Language), Java, XML (Extensible Markup Language), etc.

Attributes of WebApps:

1. Network intensiveness
2. Concurrency
3. Unpredictable load
4. Performance
5. Availability
6. Data driven
7. Content Sensitive
8. Continuous Evolution
9. Immediacy
10. Security
11. Aesthetic (Visual appearance)

SOFTWARE MYTHS

Software Myths propagated misinformation and confusion. S/w myths has a number of attributes that made them insidious; for instance, they appeared to be reasonable statements of fact, they had an intuitive feeling. Today most knowledgable professionals recognize myths for what they are - misleading attitudes that have caused serious problems for managers and technical people alike. There are two types of myths: Management and customer, and Practitioner's Myth.

1. Management Myths :- Managers with s/w responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules for slipping and improve quality. A s/w manager often grasps at belief in a s/w myth, if that belief will lessen the pressure.

Myth 1 : The standards and procedures for building s/w, won't that provide my people with everything they need to know?

Reality : It is streamlined to improve time to delivery while still maintaining a focus on quality.

Myth 2 : People have state-of-the-art s/w development tools, after all, we buy them the newest computers.

Reality : Computer Aided S/w Engg. (CASE) tools are more important than h/w for achieving good quality and productivity.

Myth 3 : If we get behind schedule, we can add more programmers and catch up.

Reality : As the new people are added, people who were working must spend time educating the new comers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well co-ordinated manner.

Myth 4 : If I decide to outsource the s/w project to the third party, I can just relax and let that firm build it.

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

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

Myth 5 : A general statement of objectives is sufficient to begin writing programs - We can fill in the details later.

Reality : It is true that s/w requirements change, but the impact of change varies with the time at which it is introduced. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential.

Myth 6 : Project requirements continually change, but change can be easily accommodated because s/w is flexible.

Reality : It is true that s/w project requirements continually change. If serious attention is given to up-front definition, early requests for change can be accommodated easily.

3. Practitioner's Myth : Myths that are still believed by s/w practitioners have been fostered by 50 years of programming culture. During the early days of s/w programming was viewed as an art form. Old ways and attitudes die hard.

Myth 7 : Once we write the program and get it to work, our job is done.

Reality : Between 60 to 80 percent of all effort expended on s/w will be expended after it is delivered to the customer for the first time.

Myth 8 : Until I get the program "running" I have no way of assessing its quality.

Reality : S/w reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of s/w defects.

Myth 9 : The only deliverable work product for a successful project is the working program.

Reality : A working program is only part of a s/w configuration that includes many elements.

Myth 10 : S/w Engg. will make us create voluminous and unnecesary documentation and invariably slow us down.

Reality : S/w Engg. is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced work results in faster delivery times.

REQUIREMENTS GATHERING & ANALYSIS

The analyst starts the requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. Then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem.

Requirements gathering : This activity typically involves interviewing the end-users and customers and studying the existing documents to collect all possible information regarding the system. However, in the absence of a working system, much more imagination and creativity on the part of the system analyst is required.

Analysis of gathered requirements :- The main purpose of this activity is to clearly understand the exact requirements of the customer. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is a problem ?
- Why is it important to solve the problem?
- What are the possible solutions ?
- What exact input, output data required ?
- What are the likely complexities?
- What exactly would the data interchange formats ?

The problems of anomalies anomalies :

a. Anomaly :- An ambiguity in the requirement.
b. Inconsistency :- might arise in the process control application.
c. Incompleteness :- Some of the requirements have overlooked.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

The SRS document usually contains all the user requirements in an informal form. Writing the SRS document is toughest. The reason behind is SRS document is expected to cater to the needs of a wide variety of audience. Some of the important categories of users of the SRS document and their needs are as follows:

a. Users, customers, and marketing personnel :- The goal of this set of audience is to ensure that the system will meet their needs.
b. S/w developers :- The s/w developers refers to the SRS document to make sure that they develop exactly what is required by the customer.
c. Test Engineers :- Their goal is to ensure that the requirements understandable from a functionality point of view, so that they

can test the s/w and and validate its working.

d. User documentation writers :- Their goal is in reading the SRS document is to ensure that they understand the document well enough to be able to write the users' manuals.

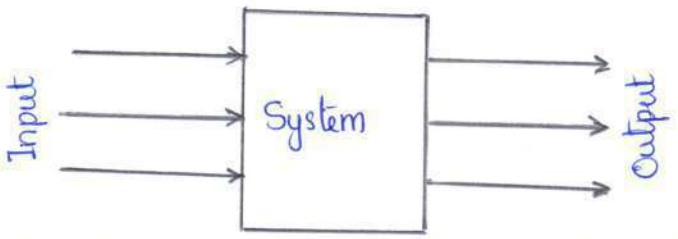
e. Project managers :- They want to ensure that they can estimate the cost easily by referring to the SRS document and that it contains all the information required to plan the project well.

f. Maintenance engineers :- The SRS document helps the maintenance engineers to understand the functionality of the system. The requirements knowledge would enable them to determine modifications.

Contents of the SRS Document :-

An SRS document should clearly document the following aspects of a system :

- Functional requirements
- Nonfunctional requirements
- Goals of implementation.



The functional requirements part should discuss the functionalities required from the system. Each function of the system (f_1) of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (O_i). The functional requirements of the system as documented in the SRS document should clearly describe each function which the system would support along with the corresponding input and output data set.

TRACEABILITY

Traceability means that it would be possible to tell which design component corresponds to which requirement, which code corresponds to which design component, and which test case corresponds to which requirement, etc. Traceability analysis is an important concept and is frequently used during s/w development. A basic requirement to achieve traceability is that each functional requirement should be numbered uniquely and consistently. Proper numbering of the requirements makes it possible to uniquely refer to specific requirements in different documents

CHARACTERISTICS OF A GOOD SRS DOCUMENT

The characteristics and the skill of writing a good SRS document usually comes from the experience gained from writing similar documents for many problems. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. Some of the identified desirable qualities of the SRS documents are the following:

Concise :- The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

Structured: The SRS document should be well-structured. A well structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

Black-box view :- The SRS document should specify the

external behavior of the system and not discuss the implementation issues. It should view the system to be developed as a black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

Conceptual Integrity :- The SRS document should characterize acceptable responses to undesired events. These are called system responses to exceptional conditions.

Response to undesired events :- The SRS document should exhibit integrity so that the reader can easily understand the contents.

Verifiable :- All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation. Requirements such as 'the system should be user friendly' are not verifiable. On the other hand, "When the name of the book is entered, the s/w should display the location of the book, if the book is available", is verifiable. Any feature of the required system that is not verifiable should be listed separately in the 'goals of the implementation' section of the SRS document.

Examples of Bad SRS Documents:

SRS document written by practitioners in industry and by students as classroom exercises, frequently suffer from a variety of problems. Some of the important categories of problems that many SRS documents suffer from are as follows:

- Over specification
- Forward references
- Wishful thinking

IEEE 830 GUIDELINES

- Help s/w customers to accurately describe what they wish to obtain.
- Help s/w suppliers to understand exactly what the customer wants.
- Help participants to -
  - Develop a template (format and content) for the s/w requirements specification (SRS) in their own organizations.
  - Develop additional documents such as SRS quality checklists or an SRS writer's handbook.
  - Establish the basis for agreement between the customers and the suppliers on what the s/w product is to do.
  - Reduce the development effort.
  - Forced to consider requirements early
  - Reduces later redesign, recoding and retesting
  - Provide a basis for realistic estimates of costs and schedules
  - Provide a basis for validation and verification
  - Facilitate transfer of the s/w product to new users or new machines.
  - Serve as a basis for enhancement requests.

REPRESENTING COMPLEX REQUIREMENTS USING DECISION TABLES & DECISION TREES

A simple text description, in cases where complex conditions would have to be checked and large number of alternatives might exist, would be difficult to comprehend and analyze. In such a situation, a decision tree or a decision table can be used to represent the logic and processing involved.

There are two main techniques available to analyze and represent complex processing logic : decision trees and decision tables. Once the decision-making logic is captured in the form of trees or tables, the test cases to validate the logic can be automatically obtained. For Instance, decision trees and decision tables find applications in information theory and switching theory.

Decision Tree :-

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. Decision table specify which variables are to be tested, and based on this what actions is to be taken depending upon the outcome of the decision-making logic, and the order in which decision making is performed.

The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition. Instead of discussing let us understand through below example:

Example :- A Library Membership Software (LMS) should support the following three options : new member, renewal and cancel membership. When the "new member" option is selected.

the s/w should ask for the member's name, address and phone number. If proper information is entered, the s/w should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable. If the "renewal" option is chosen, the LMS should ask for the member's name and the membership number. If the members details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership details entered are invalid, an error message should be displayed. If the "cancel membership" option is selected and the name of the valid member is entered, then the membership is cancelled, a cheque for the balance amount due to the member is printed and his membership record is deleted.

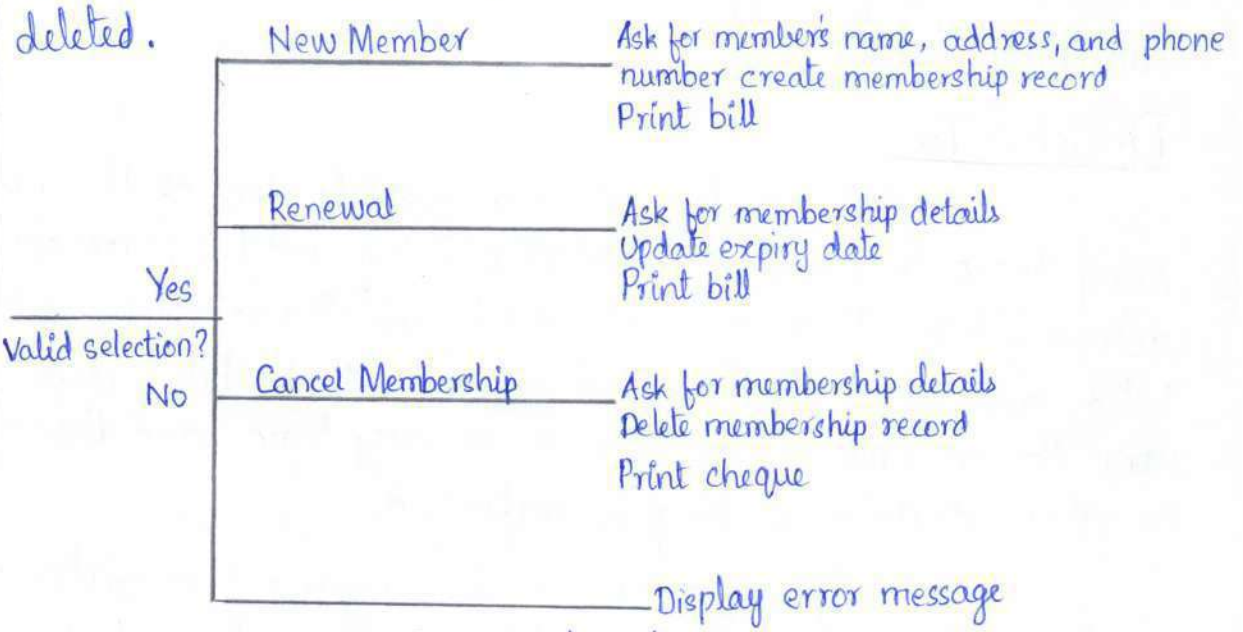


Fig: Decision tree for LMS

Decision Table

A decision table shows the decision making logic and the corresponding actions taken in a tabular or matrix form. The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an

evaluation test is satisfied. A column in the table is called a rule. A rule implies that if a condition is true, then the corresponding action is to be executed. The decision table for the LMS problem is as follows:

CONDITIONS				
Valid selection	NO	YES	YES	YES
New members	-	YES	NO	NO
Renewal	-	No	YES	NO
Cancellation	-	No	NO	YES
Actions				
Display error message	X			
Ask for member's name etc.	X			
Build customer record		X		
Generate bill		X	X	
Ask for membership details			X	X
Update expiry date			X	
Print cheque				X
Delete record				X

Even though both decision tables and decision trees can be used to represent complex program logic, decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit. A decision tree is more useful in a situation where multilevel decision making is required. Decision trees can represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.

Overview of Formal System Development Techniques

In recent years, formal techniques have emerged as a central issue in s/w engineering. This is not accidental; the importance of precise specification, modeling and verification is recognized in most engineering disciplines. Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented. The specification of a system can be given either as a list of its desirable properties or as an abstract model of the system. These two approaches are discussed, will first highlight some important concepts in formal methods, and then examine the merits and demerits of using formal techniques.

1. What is a Formal Technique :- A formal technique is a mathematical method used to: specify a h/w and/or a s/w system. The mathematical basis of formal method is provided by its specification language. More precisely, a formal specification language consists of two sets of syn and sem, and a relation sat between them. The set syn is syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn, and model of the system sem, if sat, then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

The different stages in the system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every stage of the system development activity.

Semantic Domains: Formal techniques can have different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values.

Syntactic domains :- The syntactic domain of a formal specification language consists of an alphabet of symbols and set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.

Satisfaction relation :- It is determined by using a homomorphism known as semantic abstraction function. There can be different specifications describing different aspects of a system model, possibly using different specification languages.

2. Model Vs Property-Oriented Methods :- Formal methods are usually classified into two broad categories - model-oriented and property-oriented approaches. In a model-oriented style, one defines a system behavior directly by constructing a model of the system in terms of the mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the property-oriented style, the system's behavior is defined indirectly by stating its properties, usually in the form of a set of axioms the system must satisfy.

In a model-oriented approach, we would start by defining the basic operations, p (produce) and c (consume). Then we can state that $SI + p \Rightarrow S$, $S + c \Rightarrow SI$. Thus, the model-oriented approaches essentially specify a program by writing another, presumably simpler, program.

3. Operational Semantics : The operational semantics of a formal method constitute the ways computations are represented. There are different types of operational semantics according what is means by a single run of the system and how the runs are grouped together to describe the behavior of the system. Commonly used semantics are :

- Linear Semantics :- a sequence of events or states.
- Branching Semantics : represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system.
- Maximally parallel Semantics :- all concurrent actions enabled at any

state are assumed to be taken together.

- Partial order semantics :- constitute a structure of states satisfying a partial order relation among the states or events.

Merits and Limitations of Formal Methods :-

- Formal specifications encourage rigour.
- Formal methods usually have a well founded mathematical basis.
- Formal methods have well-defined semantics.
- The mathematical basis of the formal methods facilitates automating the analysis of specifications.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system.

AXIOMATIC SPECIFICATION

In axiomatic specification of a system, the first-order logic is used to write the pre and post-conditions in order to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can be successfully invoked. In essense, the pre conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function complete execution for the function to be considered to have executed successfully. Thus, the post-conditions are essentially the constraints on the result produced for the function execution to be considered successful.

How to develop an axiomatic specification ?

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a functions.

- Establish the range of input values over which the function

should behave correctly. Establish the constraints on the input parameters as a predicate.

- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Continue all of the above into pre and post-conditions of the function.

Example 1: Specify the pre and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$$f(x : real) : real$$
$$pre : x \in R$$
$$post : \{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2 * x)\}$$

Example 2: Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

$$Search(X : IntArray, key : Integer) : Integer$$
$$pre : \exists i \in [X first \ldots X last], X[i] = key$$
$$post : \{(X'[Search(X, key)] = key) \wedge (X = X')\}$$

ALGEBRAIC SPECIFICATION

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag, in the specification of abstract data type. Algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations $(I, +, -, *, /)$. In contrast

alphabetic strings together with operations of concatenation and length (A, 1, Con, len), do not form a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in the algebra, in turn, is called a sort of the algebra. Using algebraic specification, we define the meaning of a set of interface procedures by using equations. An algebraic specification is usually presented in four sections.

1. Types section :- In this section, the sorts (or the data types) being used are specified.
2. Exception Section :- This section gives the names of the exceptional conditions that might occur when different operations are carried out.
3. Syntax section :- This section defines the signatures of the interface procedures. The collection of sets that form the input domain of an operator and the sort where the output is produced are called the signature of the operator. For example, PUSH takes a stack and an element and returns a new stack.
4. Equation Section :- This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

The first step in defining an algebraic specification is to identify the set of required operations. After identified the required operators, it is helpful to classify them as either basic constructors, extra constructors, basic inspectors, or extra inspectors. The definition of these categories of operators are as follows:

1. Basic construction operators :- These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible elements of the type being specified. For example, 'create' and 'append' are basic construction operators.

2. Extra Construction Operators :- These are the construction operators other than the basic construction operators. For example, the operator 'remove' is an extra construction operator, because even without using 'remove' it is possible to generate all values of the type being specified.

3. Basic inspection operators :- These operators evaluate attributes of a type without modifying them. e.g. eval, get etc.. Let S be the set of operators whose range is not the data type being specified. The set of the basic operators S1 is a subset of S, such that each operator from S-S1 can be expressed in terms of the operators from S1.

4. Extra Inspection operators :- These are the inspection operators that are not basic inspectors.

A simple way to determine whether an operator is a constructor or an inspector is to check the syntax expression for the operator. A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor and inspection operators. Then write an axiom for the composition of each basic construction operator over each basic inspection operator and extra constructor operator.

Thus, if there are m1 basic constructors, m2 extra constructors, n1 basic instructors inspectors, and n2 extra inspectors, we should have $m1 \times (m2 + n1) + n2$ axioms. However, it should be clearly noted that these $m1 \times (m2 + n1) + n2$ axioms are the minimum required and many more axioms may be needed to take the specification complete.

While developing the rewrite rules, different persons can come up with different sets of equations. However, while developing the equations one has to be careful to see that the equations are able

to handle all meaningful composition of operators, and they should have the unique termination and finite termination properties.

Example :- Let us specify a data type point supporting the operations create, xcoord, ycoord, isequal, where the operations have their usual meaning.

Types :
defines point
uses boolean, integer

Syntax :-
1. create : integer x integer → point
2. xcoord : point → integer
3. ycoord : point → integer
4. isequal : point x point → boolean

Equations :
1. xcoord (create (x,y)) = x
2. ycoord (create (x,y)) = y
3. isequal (create (x1, y1), create (x2, y2)) = ((x1 = x2) and (y1 = y2))

In this example, we have only one basic constructor (create), and three basic inspectors (xcoord, ycoord, isequal). Therefore, we have only three equations.

The rewrite rules let you determine the meaning of any sequence of calls on the point type. Consider the following expression:

isequal (create (xcoord (create (2,3)), 5), create (ycoord (create (2,3)), 5).

By applying the rewrite rule 1 you can specify the given expression as

isequal (create (2,5), create (ycoord (create (2,3)), 5)).

By using the rewrite rule 2, you can further simplify this as

isequal (create (2,5), create (3,5)).

This is false by rewrite rule 3.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are :

- Completeness. This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.
- Finite termination property. This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right-hand side of each rewrite rule has fewer rule has terms than the left, then the rewrite process must terminate.
- Unique termination property : This property indicates whether application of rewrite rules in different orders always results in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked : can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique terminations property is a very difficult problem.

Example :- Let us specify a FIFO queue supporting the operations create, append, remove, first and isempty, where the operations have their usual meaning.

Types :
    defines queue
    uses boolean, element
Exception :
    underflow, novalue
Syntax :
1. create : $\phi \rightarrow$ queue

2. append : queue x element → queue
3. remove : queue → queue + (underflow)
4. first : queue → element + (novalue)
5. isempty : queue → boolean

Equations :

1. isempty (create()) = true
2. isempty (append (q, e)) = false
3. first (create()) = novalue
4. first (append (q, e)) = if isempty (q) then e else first (q)
5. remove (create()) = underflow
6. remove (append (q, e)) = if isempty (q) then create() else append (remove (q), e)

In example, we have two basic construction operators (create and append). We have one extra construction operator (remove). We have considered remove to be an extra construction operator because all values of the queue can be realized, even without having the remove operator. We have two basic inspectors (first and isempty). Therefore, we have 2 x 3 = 6 equations.

—•• END OF UNIT-II ••—

UNIT - III

SOFTWARE DESIGN - Good Software Design, Cohesion and Coupling, Control Hierarchy : Layering, control abstraction, Depth and Width, Fan-out, Fan-in, Software design approaches, object oriented vs function oriented design. Overview of SA/SD methodology, structured analysis, Data flow diagram, Extending DFD technique to real life systems, Basic Object Oriented concepts, UML Diagrams, Structured design, Detailed design, Design review, Characteristics of a good user interface, User Guidance and Online Help, Mode-based Vs Mode-less Interface, Types of user interfaces, Component-based GUI development, User Interface design methodology : GUI design methodology.

SOFTWARE DESIGN :

S/w design deals with transforming the customer requirements into a form that is implementable using a programming language. For a design to be easily in a conventional programming language, the following items must be designed during the phase:

- Different modules required to implement the design solution.
- Control relationship among the identified modules
- Interface among different modules
- Data structures of the individual modules
- Algorithms required to implement the individual modules.

A good s/w design is seldom achieved by using a single-step procedure but requires several iterations.

GOOD SOFTWARE DESIGN

The definition of a good s/w design can vary depending on the application for which it is being designed. For example, the memory size used up by a program may be an

important issue to characterize a good solution for embedded s/w development - since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption. The goodness of a design is dependent on the targetted application. The notation of goodness of a design itself varies widely across s/w engineers and academicians. The characteristics of good design are listed below:

- Correctness: A good design should correctly implement all the functionalities of the system.
- Understandability: A good design should be easily understandable.
- Efficiency : It should be efficient.
- Maintainability: It should be easily amenable to change.

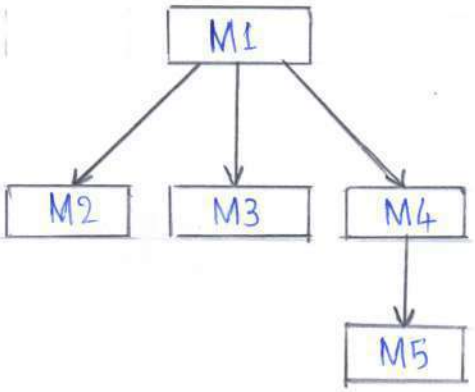
A design has to be correct to be acceptable. A design that is easy to understand is also easy to develop, maintain, and change. Thus, unless a design is easily understandable, it would require tremendous effort to implement and maintain it. In order to facilitate understandability of a design, the design should have the following features:

- It should use consistent and meaningful names for various design components.
- The design should be modular.
- It should neatly arrange the modules in a hierarchy, e.g. tree-like diagram.

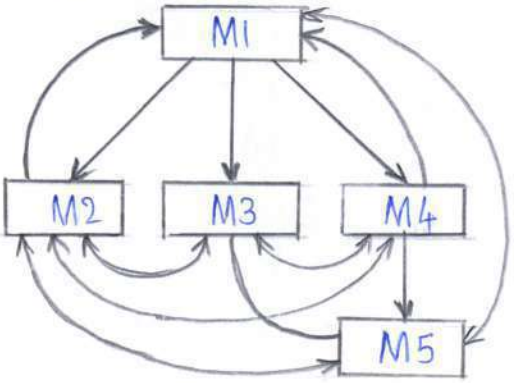
Modularity : Decomposition of a problem into modules facilitates the design by taking advantage of the divide and conquer principle. The modularity reduces the complexity of the design solution greatly.

Clean decomposition :- It means that in a s/w design should display

high cohesion and low coupling, Conceptually it means that the modules are more or less independent of each other.



(a) Modular design



(b) Design exhibiting poor modularity

For example, of the two design solutions represented in the above figures, it is obvious that the design solution of Figure (a) is easier to understand.

Neat Arrangement : This aspect is essentially characterized by the following:

- Layered solution
- Low fan-out and
- Abstraction

COHESION AND COUPLING

The primary characteristics of a neat module decomposition are high cohesion and low coupling. Cho Cohesion is a measure of the functional strength of a module ; where as the coupling between two modules is a measure of the degree of independence or interaction between the two modules.

A module having high cohesion and low coupling is said to be functionally independent of other modules. The functional independence means that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules. Functional independence is a key to any good

design primarily due to the following reasons.

Error isolation : Any error in a module would not directly affect the other modules.

Scope for reuse : A cohesive module can be easily taken out and be reused in a different program.

Understandability : Complexity of the design is reduced, because different modules are more or less independent of each other and can be understood in isolation.

Classification of Cohesiveness :

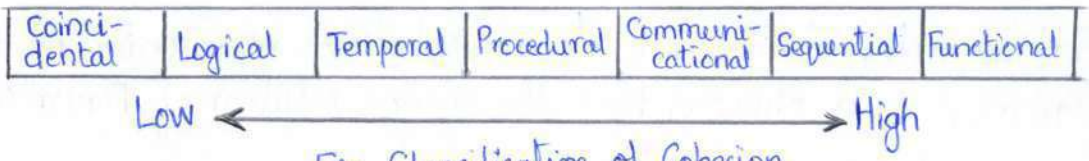


Fig. Classification of Cohesion

The different classes of cohesion that a module may possess are depicted in above figure. The classes of cohesion are elaborated below.

Coincidental Cohesion : It performs a set of tasks that relate to each other very loosely. In this case, the module contains a random collection of functions.

Logical Cohesion : If all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

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, shut-down of some process, etc. exhibit temporal cohesion.

Procedural Cohesion : If the set of functions of module are all part of a procedure (algorithm) in which certain sequence of steps has

to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

Communicational Cohesion :- If all the functions of the module refer to or update the same data structure, e.g. the set of functions define on an array or a stack.

Sequential Cohesion :- If the elements of a module form the parts of a sequence, where the output from one element of the sequence is input to the next.

Functional Cohesion :- If the different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' payroll displays functional cohesion.

CLASSIFICATION OF COUPLING

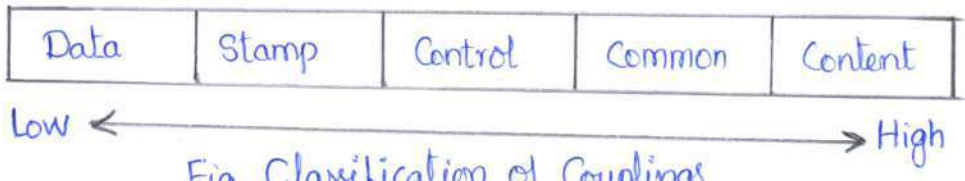


Fig. Classification of Couplings

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Let us now classify the different types of couplings that can exist between different modules. Five types of couplings can occur between any two modules.

Data Coupling :- Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter

between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

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 instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

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

Content Coupling :- Content coupling exists between two modules, if their code is shared, e.g. branch from one module into another module.

The degree of coupling is in ascending order from data coupling to content coupling. High coupling among modules not only makes a design difficult to understand and maintain, but it also increases development effort as the modules having high coupling cannot be developed independently by different team members. Modules having high coupling are difficult to implement and debug.

CONTROL HIERARCHY

The control hierarchy represents the organization of the program components. The control hierarchy is also called program structure. Many different types of notations have been used to represent the control hierarchy. The most common notation is a tree-like diagram known as structure chart. However, other notations such as Warnier-Orr or Jackson diagrams may also be used. Since, Warrier-Orr or Jackson's notation are not widely

FAN-OUT

It is a measure of number of modules that are directly controlled by a given module. A design having modules with high fan-out numbers is not a good design as such modules would lack cohesion. This is due to the fact that a module having a large fan-out number invokes a large number of other modules and is likely to implement several different functions and not just a single cohesive function.

FAN-IN

It indicates the number of modules directly invoking a given module. High fan-in represents code reuse and is, in general, encouraged.

SOFTWARE DESIGN APPROACHES

There are fundamentally two different approaches to s/w design - function-oriented design, and object-oriented design. Each of these two techniques may be applicable at different stages of the design process.

OBJECT-ORIENTED Vs. FUNCTION-ORIENTED DESIGN

The following are some of the important differences between function-oriented and Object-oriented design.

- Unlike function-oriented design methods, in OOD the basic abstractions are not real-world functions such as sort, display, track, etc. but real-world entities such as employee, picture, machine radar system, etc.
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. Therefore, one object may discover the state information of another object by interrogating it.

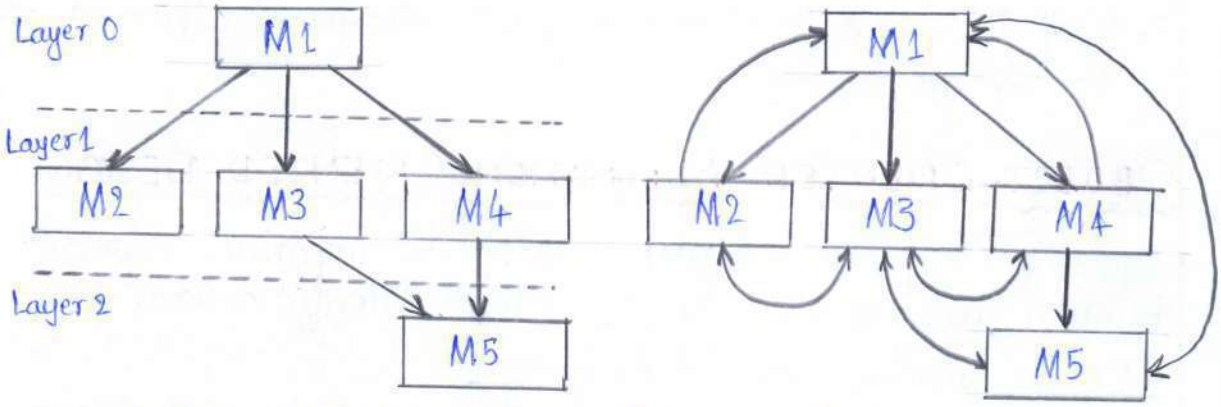
used now a days.

LAYERING

In a layered design solution, the modules are arranged in layers. A module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller. The control hierarchy also represents a subtle characteristic of s/w architecture i.e. its visibility. A module A is said to be visible to another module B, if A directly or indirectly calls B.

CONTROL ABSTRACTION

A module should invoke the functions of the modules in the layer immediately below it. A module at a lower layer, should not invoke the services of the modules above it. In other words, the modules at the higher layers, should not be visible at the modules at the lower layers.



(a) Layered design with good control abstraction

(b) Layered design showing poor control abstraction

Fig. Examples of good and poor abstraction

DEPTH AND WIDTH

These provide an indication of the number of levels of control and the overall span of control respectively.

- Function-oriented techniques, such as SA/SD, group the functions together if, as a group, they constitute a higher-level function. On the other hand, the object-oriented techniques group the functions together on the basis of the data they operate on.

To illustrate the differences between the Object-oriented and the function-oriented design approaches, let us consider an example- that of an automated fire-alarm system for a large building.

Fire-alarm system:

The owner of a large multistoried building wants to have a computerized fire-alarm system for building. Smoke detectors and fire-alarm sirens would be placed in each room of the building. The fire-alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire-alarm system should determine the location at which the fire condition has occurred and then sound the sirens only in the neighbouring locations. The fire-alarm system should also flash an alarm message on the computer console. Fire fighting personnel console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the sirens by the fire fighting personnel.

Function-oriented approach

```
/* Global data (system state) accessible by various functions */
    BOOL detector_status [MAX_ROOMS];
    int detector_locs [MAX_ROOMS];
    BOOL alarm_status [MAX_ROOMS]; /* alarm activated when status is set */
    int alarm_locs [MAX_ROOMS]; /* room number where alarm is set */
    int neighbour_alarms [MAX_ROOMS] [10]; /* each detector has at most */
                                           /* 10 neighbouring alarm locations */
```

The functions which operate on the system state are:

```
interrogate_detectors();
get_detector_location();
```

```
determine_neighbour();
ring-alarm();
reset_alarm();
report_fire_location();
```

Object-oriented approach

class detector
attributes: status, location, neighbours
operations: create, sense-status, get-location, find-neighbours
class alarm
attributes: location, status
operations: create, ring-alarm, get-location, reset-alarm

Even though object-oriented and function-oriented approaches are remarkably different approaches to s/w design, they do complement each other in some sense. For example, usually one applies the top-down function-oriented techniques to design the internal methods of a class, once the classes are identified. In this case outwardly the system appears to have been developed in an object-oriented fashion, inside each class there may be a small hierarchy of functions in a top-down manner.

OVERVIEW OF SA/SD METHODOLOGY

The SA/SD technique can be used to perform the high-level design of a s/w. As the name implies, the SA/SD methodology consists of two different activities:

- Structured Analysis (SA)
- Structured Design (SD)

The aim of the structured analysis activity is to transform a textual problem description into a graphic model. More precisely, structured analysis is used to carry out the top-down decomposition of the set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional

decomposition of the system is achieved. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called s/w architecture for the given problem and it can be directly implemented using a conventional programming language.

STRUCTURED ANALYSIS

Structure Analysis technique is based on the following essential underlying principles:

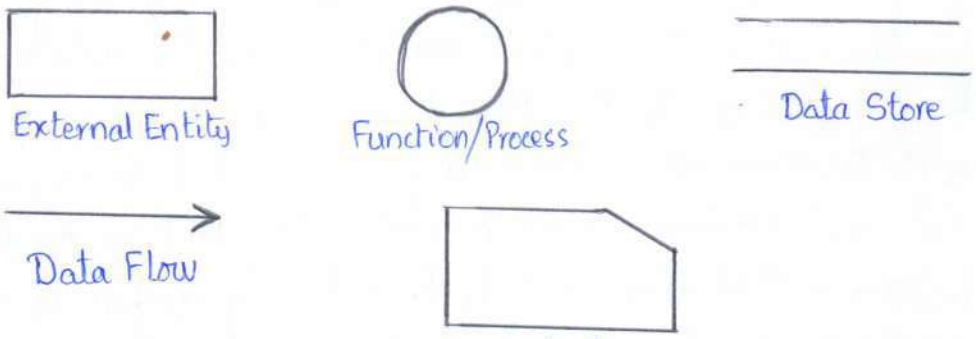
- Top-down decomposition approach
- Divide and conquer. Each function is decomposed independently
- Graphical representation of the analysis results using DFDs.

A DFD, in simple words, in 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. In the DFD terminology, it is useful to consider each function as a processing station that consumes some input data and produces some output data. DFD is an elegant modeling technique that turns out to be not only useful to represent the result of structured analysis of a s/w problem but also useful for several other applications such as showing the flow of documents or items in an organization.

DATA FLOW DIAGRAMS (DFDs)

The DFD (Bubble Chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data, and the output data is generated by the system. The main reason why the DFD technique is popular because of the fact that DFD is a very simple formalism - it is simple to understand and use. A DFD model uses a very limited number of primitive symbols to represent the functions performed by a system and the data flow among these functions. The DFD technique also follows a very simple set of intuitive concepts and rules.

Primitive symbols used for constructing DFDs



There are five different types of primitive symbols used for constructing DFDs. The meaning of each of these symbols explaind below:

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

External Entity symbol :- These are essentially those physical enlities to the s/w system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external enlity symbols can be used to represent external h/w and s/w such as another application s/w.

Data flow symbol :- A directed arc or an arrow is used as a data flow symbol. This 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 flow symbols are usually annotated with the corresponding data names.

Data Store symbol :- A data store represents a logical file. It is represented using two parallel lines. A logical file can represent either a data store symbol which can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence arrows connecting to a data store need not be annotated with the name of the corresponding data items.

Output symbol :- The output symbol is used when a hard copy is produced and the user of the copies cannot be clearly specified or there are several users of the output.

Data Dictionary : A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in the DFD model of a system. It may be recalled that the DFD model of a system typically consists of several DFDs, namely level 0 DFD, level 1 DFD, level 2 DFD, etc. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.

Data definition :- Composite data items can be defined in terms of primitive data items using the following data definition operators.

+ : composition of two data items e.g. a+b

[,,] : represents selection

( ) : optional data which may or may not appear.
{} : represents iterative data definition
= : represents equivalence
/* : represents a comment

Context diagram :- The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labelled according to the main function of the system. The various external entities with which the system interacts and the data flows occurring between the system and the external entities are also represented. To develop the context diagram of the system, we have to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data system. Here, the term users of the system also includes the external systems which supply data to or receive data from the system.

The bubble in the context diagram is annotated with the name of the S/w system being developed. This is in contrast with the bubbles in all other levels which are annotated with verbs. This is to be expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.

Example : Tic-Tac-Toe Computer Game

Tic-Tac-Toe is a computer game in which a human player and the computer make alternate moves on a 3×3 square. A move consists of making a previously unmarked square. The player who is first to place three consecutive marks along a straight line on the square wins. As soon as either the human player or the computer wins, a message congratulating the winner is displayed. If neither player messages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The context diagram and the level 1 DFD are shown in the figure.

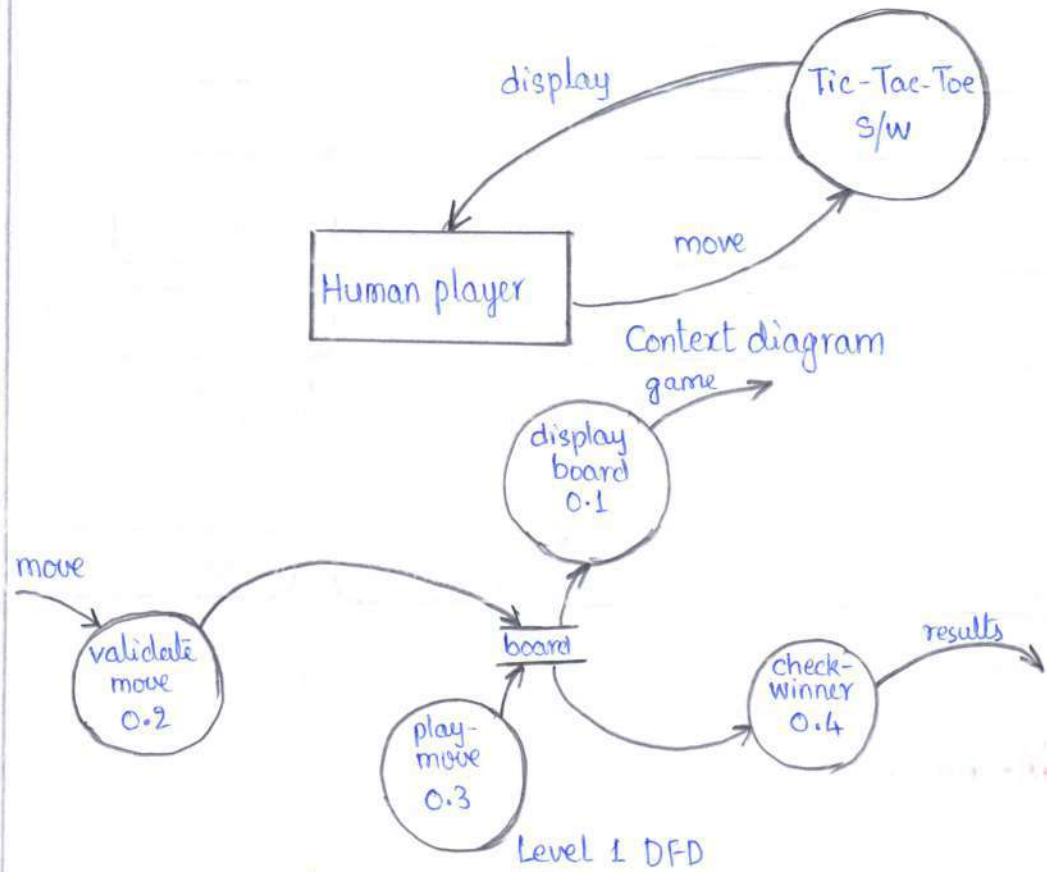


Fig: Context diagram and level 1 DFDs

Data dictionary for the DFD model :

move : integer /* number between 1 to 9 */
display : game + result
game : board
board : (integer) 9
result : ["Computer won", "human won", "drawn"]

BASIC OBJECT ORIENTED CONCEPTS

Some important concepts and terms related to the object-oriented approach are pictorially shown in figure. The basic mechanisms namely objects, classes, inheritance, messages and methods.

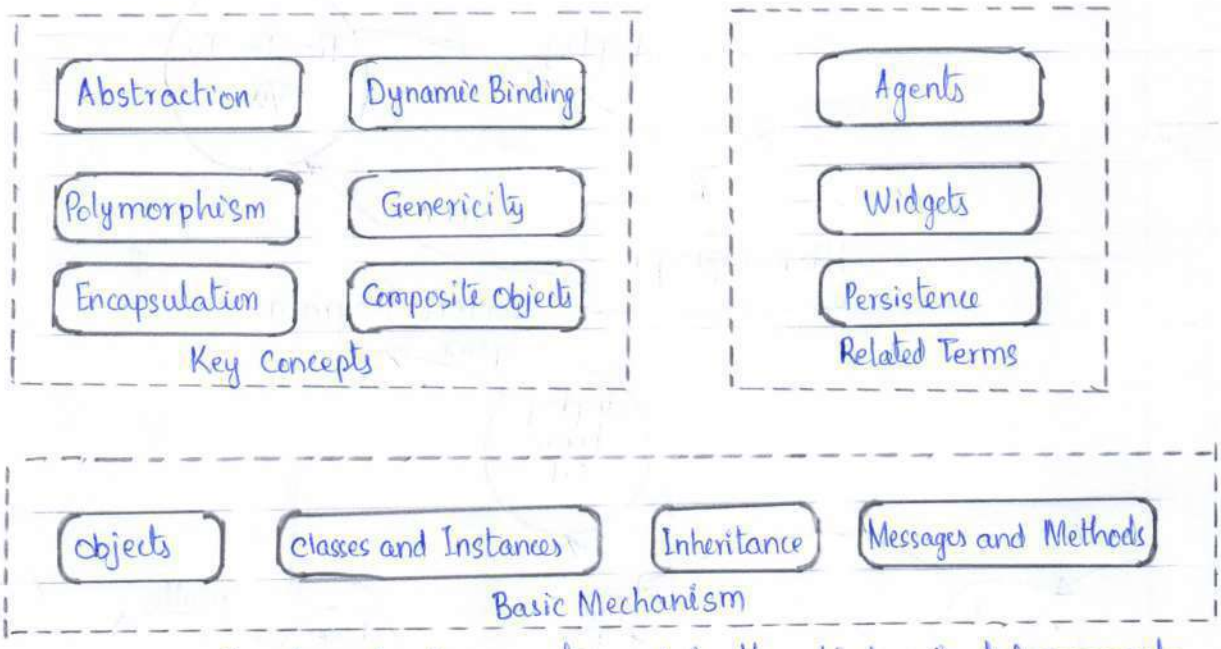


Fig: Important concepts used in the object-oriented approach

Basic Mechanisms

The following are some of the basic mechanisms used in the object-oriented paradigm.

Objects : In the object-oriented approach, a system is designed as a set of interacting objects. Normally, each object represents a tangible real-world entity such as a library member, an employee, a book, etc.

Class : Similar objects constitute a class. This means, objects possessing similar attributes and displaying similar behavior constitute a class. Once we define a class it serves as a template for object creation. Since, each object is created as an instance of some class, classes can be considered as abstract data types.

Methods and Messages : The operations supported by an object are called its methods. Thus operations and methods are almost identical terms, except for a minor technical difference as explained in the context of polymorphism. Methods are the only means available to other objects for accessing and manipulating the data of another object. The methods of an object are invoked by sending messages to it. The set of valid messages to an object constitutes its protocol.

Inheritance :- The inheritance feature allows to define a new class by extending or modifying an existing class. The original class is called the base class (or superclass) and new class obtained through inheritance is called the derived class (or sub class). A base class is a generalization of its derived classes. This means that the base class contains only those properties that are common to all the derived classes. Using the inheritance relationship, different classes can be arranged in a class hierarchy.

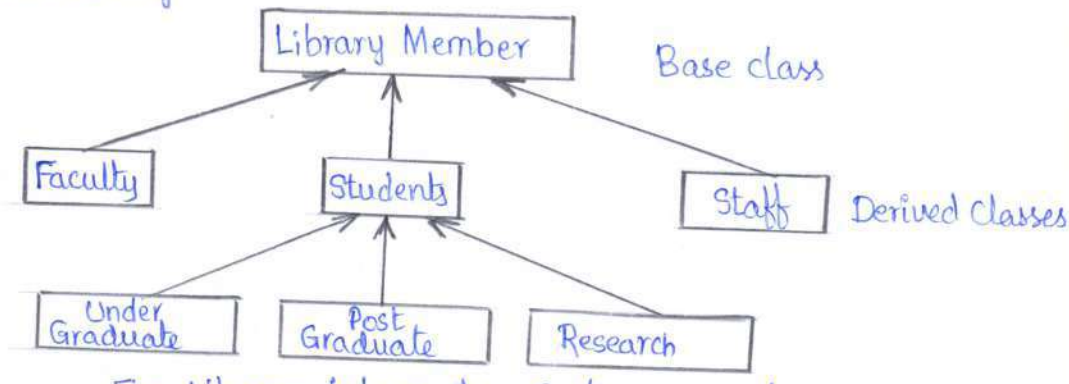


Fig. Library information System example

Multiple Inheritance :- Multiple Inheritance is a mechanism by which a subclass can inherit attributes and methods from more than one base class. Multiple inheritance is represented by arrows drawn from the subclass to each of the base classes.

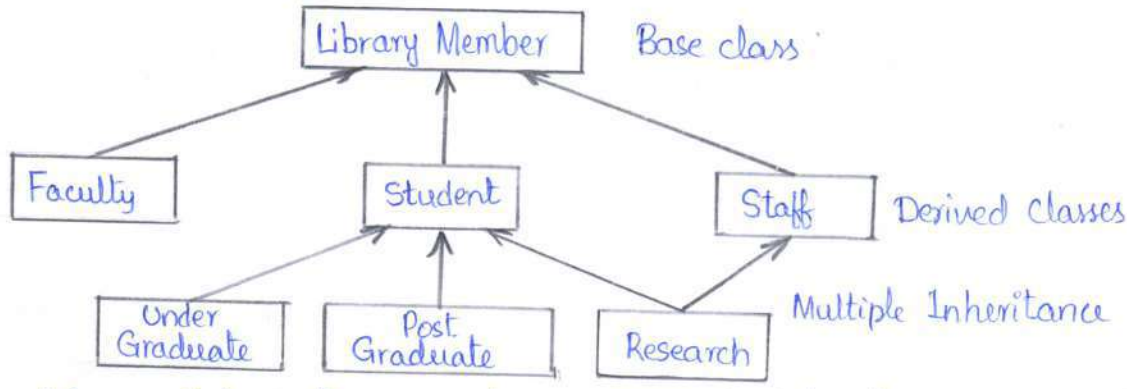


Fig. Library Info. System example with multiple Inheritance

Abstract class :- Classes that are not intended to produce instance of themselves are called abstract classes. Abstract classes merely exist so that behavior common to a variety or classes can be factored into one common location, where they can be defined once. Abstract class usually support generic methods, but the subclasses of the abstract classes are expected to provide specific implementations of these methods.

Key Concepts :

Abstraction :- Abstraction is the selective examination of certain aspects of certain aspects of a problem while ignoring the remaining aspects of the problem. Thus, the abstraction mechanism allows us to represent a problem in a simpler way by omitting unimportant details. Abstraction is a powerful mechanism for reducing complexity of a s/w. It is also viewed as a way of increasing s/w productivity.

Encapsulation :- The property of an object by which it interfaces with the outside world only through messages is referred to as encapsulation. The data of an object are encapsulated within its methods and are available only through message-based communication. Encapsulation offers three important advantages :

- It protects an object's variables from corruption by other objects.
- It hides the internal structure of an object.
- Since objects communicate among each other using messages only, they are weakly coupled.

Polymorphism :- Polymorphism literally means poly (many) morphism (forms). Polymorphism denotes the following :

- The same message can result in different actions when received by different objects. This is also referred to as static binding.
- The exact method to which a method call would be bound cannot be known at compile time, and is dynamically decided at the runtime, this is also known as dynamic binding.

Composite Objects :- Objects which contain other objects are called composite objects. Containment may be achieved by including the pointed to one object as a value in another object or by creating instances of the component objects in the composite object.

Generecity :- Generecity is the ability to parameterize class definitions. For example, while defining a class stack of different types of elements such as integer stack, character stack, and floating stack, generecity permits to define a generic class of type stack and

later instantiated it either as an integer stack, a character stack, or a floating point stack as may be required. This can be achieved by assigning a suitable value to a parameter used in the generic class definition.

Dynamic Binding :- In method invocation, static binding is said to occur if the address of the called method is known at compile time. In dynamic binding, the address of an invoked method is known only at the run time. It is useful for implementing a type of polymorphism. It is important to remember that dynamic binding occurs when we have some methods of the base class overridden by the derived classes and the instances of the derived classes are stored in instances of the base class.

Advantages of OOD :-

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability

UML DIAGRAMS

UML stands for Unified Modelling Language, it is a modelling language. It provides a set of notations e.g. rectangles, lines, ellipses, etc. to create models of systems. Models are very useful in documenting the design and analysis results. Models also facilitate the generation of analysis and design procedures themselves. UML was developed to standardize the large number of object-oriented modelling notations that existed and were used extensively in the early 1990s. The principal ones in use were:

- OMT [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- OOSE [Jacobson 1992]
- Odell's methodology [Odell 1992]
- Shlaer and Mellor methodology [Shlaer 1992]

UML was adopted by Object Management Group (OMG) as a de facto standard in 1997. UML contains an extensive set of notations and suggests construction of many types of diagrams. UML has successfully been used to model both large and small problems. Many of the UML notations are difficult to draw on paper and are best drawn using a CASE tool such as Rational Rose.

UML Diagrams:

UML can be used to construct nine different types of diagrams to capture different views of a system. The different UML diagrams provide different perspectives of the s/w system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following views of a system:

- Structural View
- Behavioral View
- Implementation view
- Environmental view

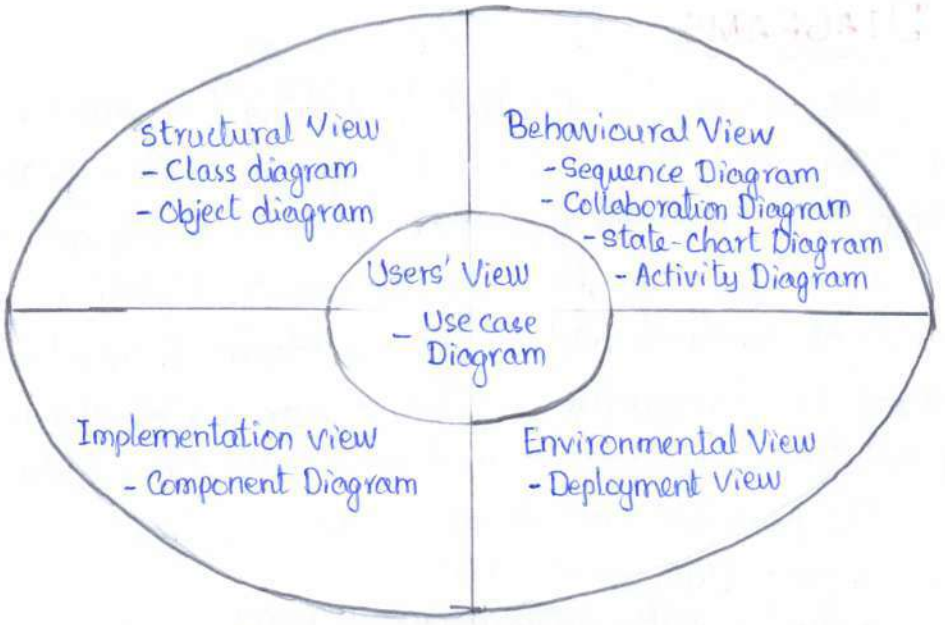


Fig: Different types of diagrams and views supported in UML

Users' View :- This view defines the functionalities made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. It is a black-box view of the system where the internal structure, the dynamic behaviour of the different system components, the implementation, etc. are not visible.

Structural View : This view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes.

Behavioural View : This view captures how objects interact with each other to realize the system behaviour. The system behaviour captures the time-dependent behavior of the system.

Implementation View : This view captures the important components of the system and their dependencies.

Environmental View : This view models how the different components are implemented on different pieces of h/w.

Use Case Model

The use case model for any system consists of a set of "use cases". Intuitively, use cases represent the different ways in which a system can be used by the users. Thus for the Supermarket prize scheme, the use cases could be:

- register customer
- register sales
- Select winners

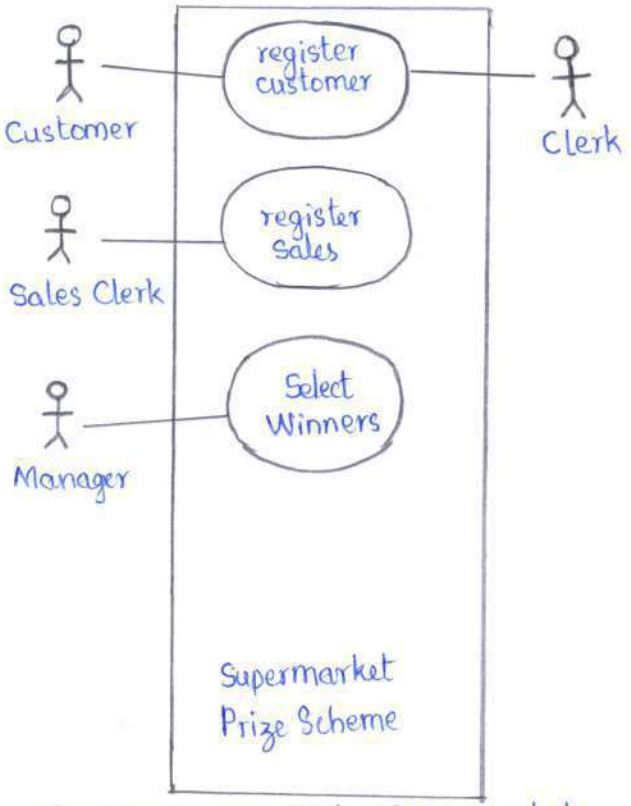


Fig: Use Case model for Supermarket Prize Scheme

Class Diagram:

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system consists of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

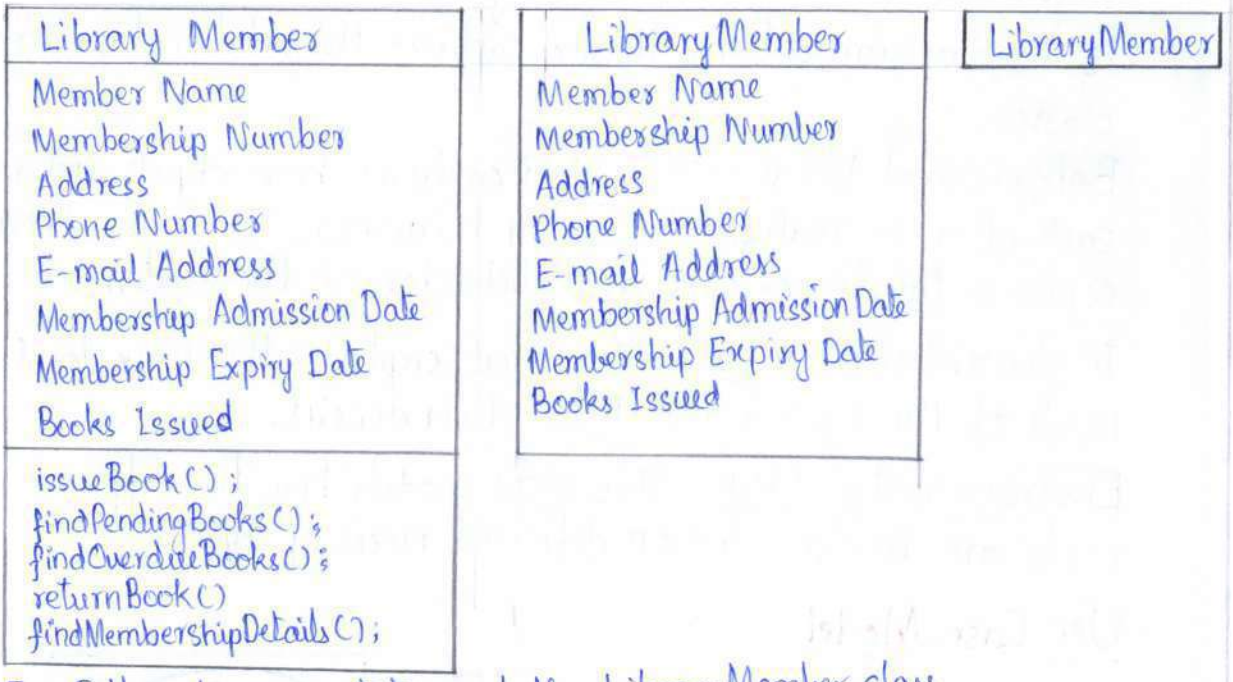


Fig: Different representations of the LibraryMember class

Interaction Diagrams

Interaction diagrams are models that describe how groups of objects collaborate to realize some behaviour. Typically, each interaction diagram realizes the behavior of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other.

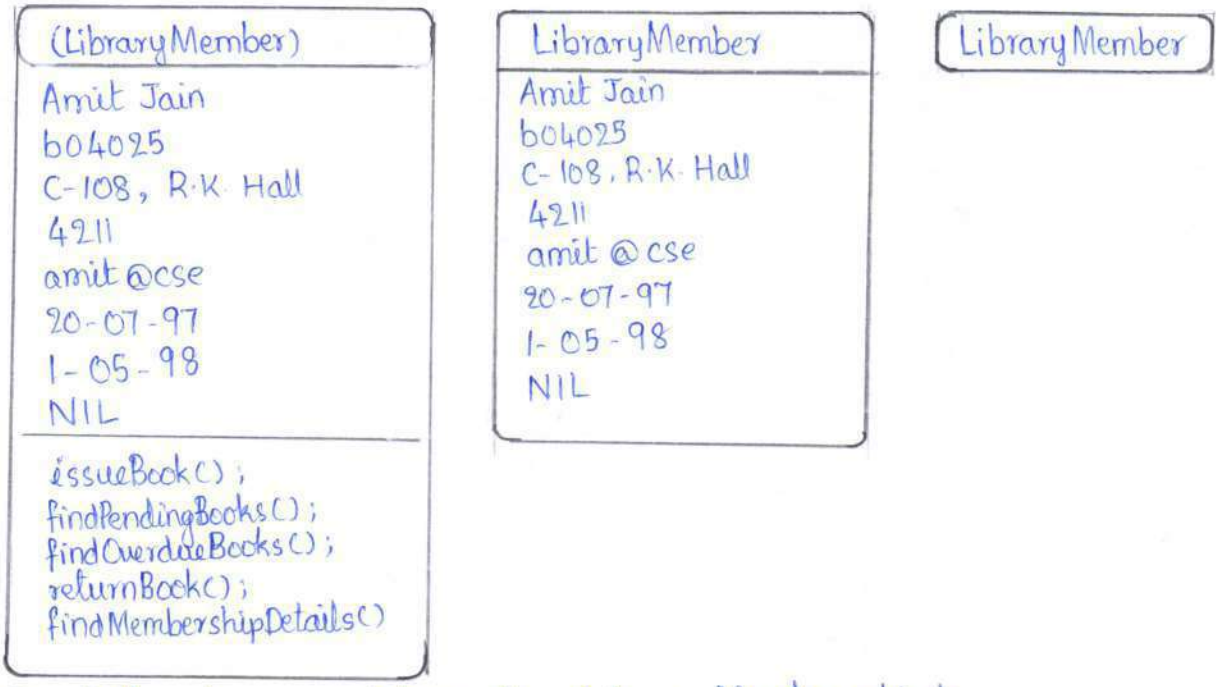


Fig: Different representations of a Library Member object

Sequence Diagram

A sequence diagram shows interaction among objects as a two-dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class, and both the name of the object and the class are underlined.

The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on lifeline is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifelines of two objects. The messages are shown in chronological order from top to bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur.

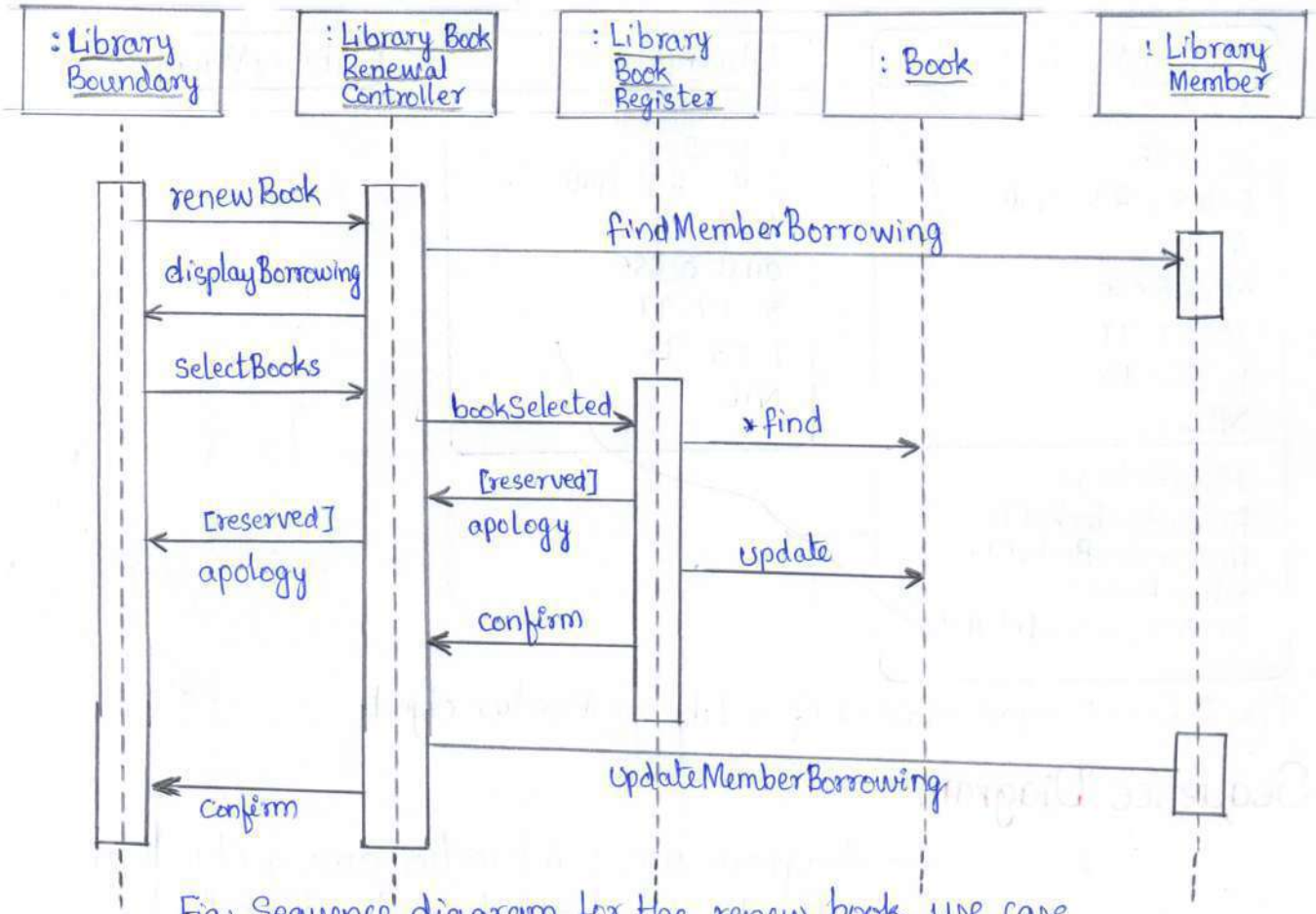


Fig: Sequence diagram for the renew book use case

Collaboration Diagram

A Collaboration diagram shows both the structural and behavioural aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects. The structural aspect of a collaboration diagram consists of objects and the link existing between them. In the diagram, an object is also called as collaborator. The behavioural aspect is described by the set of messages exchanged among the different collaborators.

The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labelled arrow placed near the link. Messages are prefixed with sequence numbers because that is the only way to describe the relative sequencing of the messages in the diagram.

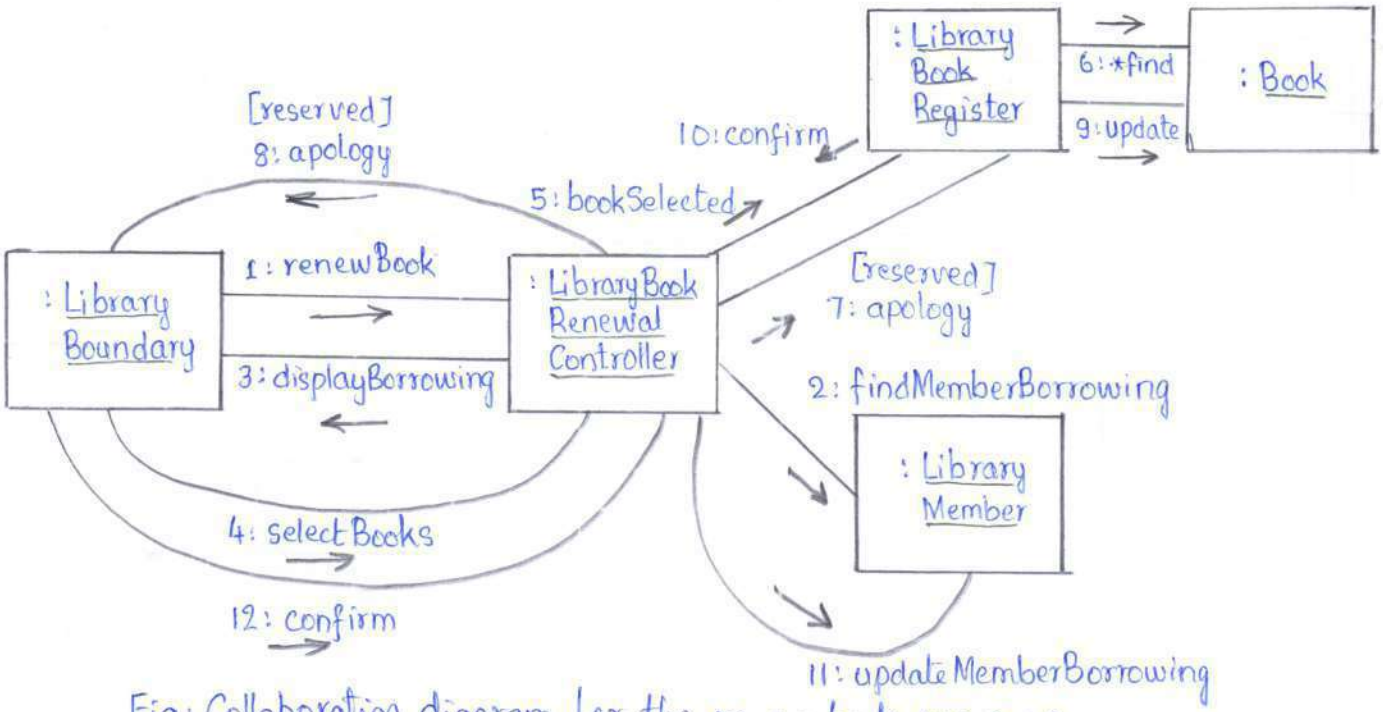


Fig: Collaboration diagram for the renew book use case

ACTIVITY DIAGRAMS

The activity diagram is possibly one modelling element which was not present in any of the predecessors of UML. It focuses on representing activities or chunks of processing which may or may not correspond to the methods of classes. An activity is a state which an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then these must be identified through conditions.

Activity diagrams are similar to the procedural flow charts. The difference is that activity diagrams support descriptions of parallel activities and synchronization aspects involved in different activities.

An interesting feature of the activity diagrams is the swim lanes. Swim lanes enables to group activities based on who is performing them. Activity diagrams are normally employed in business process modelling.

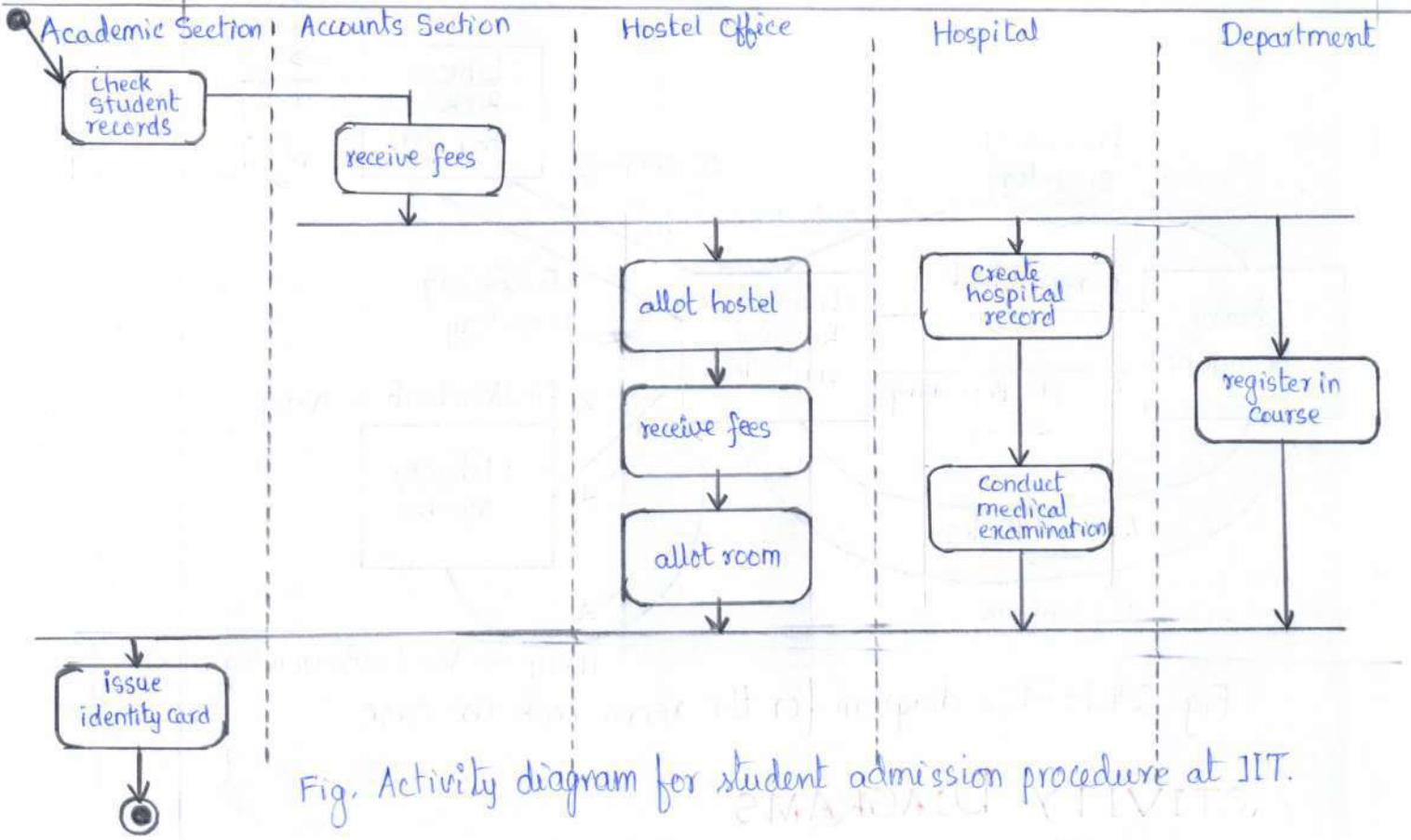


Fig. Activity diagram for student admission procedure at IIT.

State Chart Diagram

A state chart diagram is normally used to model how the state of an object changes in its lifetime. These diagrams are good at describing how the behaviour of an object changes across several use case executions.

State Chart diagrams are based on the finite state machine (FSM) formalism. An FSM consists of a finite number of states corresponding to those of the object being modelled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology, and has since been used for a wide variety of applications.

A major disadvantage of the FSM formalism is the state explosion problem. This problem is overcome in UML by using state charts.

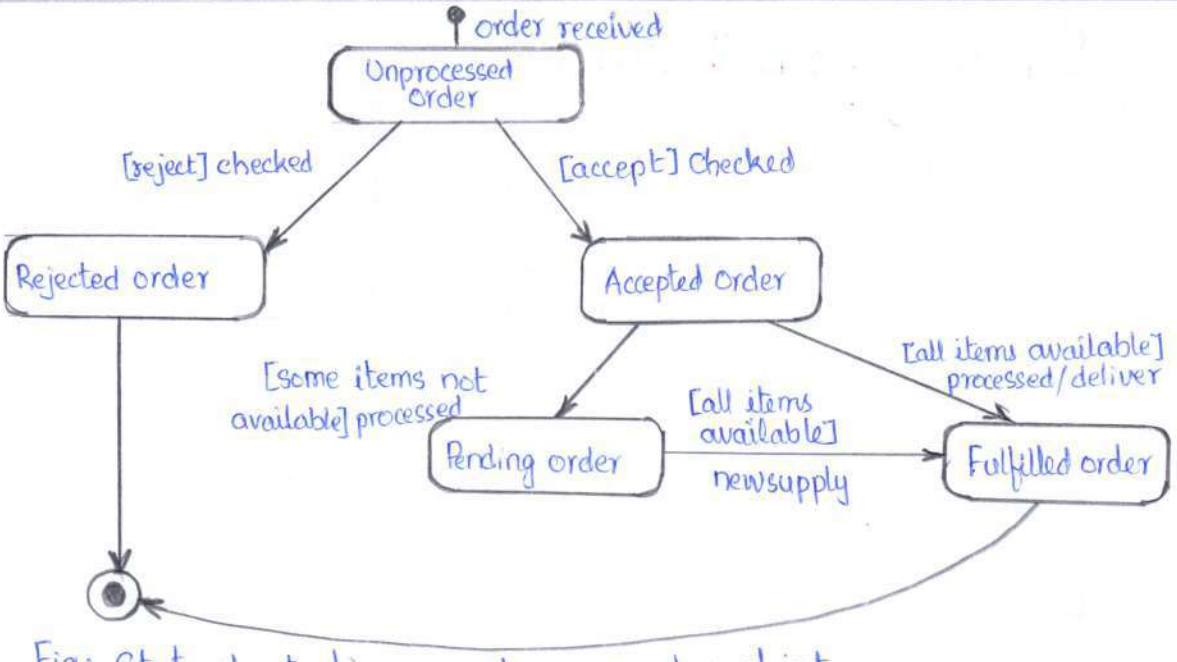


Fig: State chart diagram for an order object

The basic elements of the state chart diagram are as follows:

Initial state : It represented as a filled circle.
Final State : It is represented by a filled circle inside a larger circle.
State : It is represented by a rectangle with rounded corners.
Transition : A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is placed along side the arrow. We can also assign a guard to the transition A guard is a boolean logic condition. The transition can take place only if the grade evaluates to true. The syntax for the label of the transition is shown in three parts : event [guard]/action.

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. A structure chart represents the s/w architecture i.e. the various modules making up the system, the module dependency, and the parameters that are passed among the different modules.

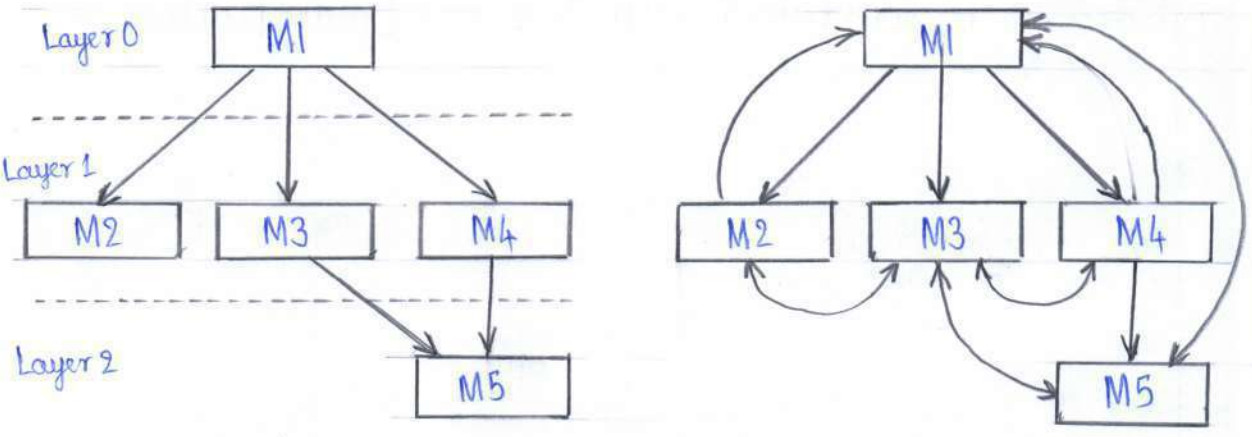
The basic building blocks which are used to design structure charts are the following:

Rectangular boxes: A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.

Module invocation arrows: An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow.

Data flow arrows: These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name.

Library modules: A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules.



(a) A properly layered design (b) A poorly layered design

Fig: Examples of properly and poorly layered design

DETAILED DESIGN

During detail design the pseudo code description of the processing and the different data structures are designed for different modules of the structure chart. These are usually written in the form of module specifications (MSPEC).

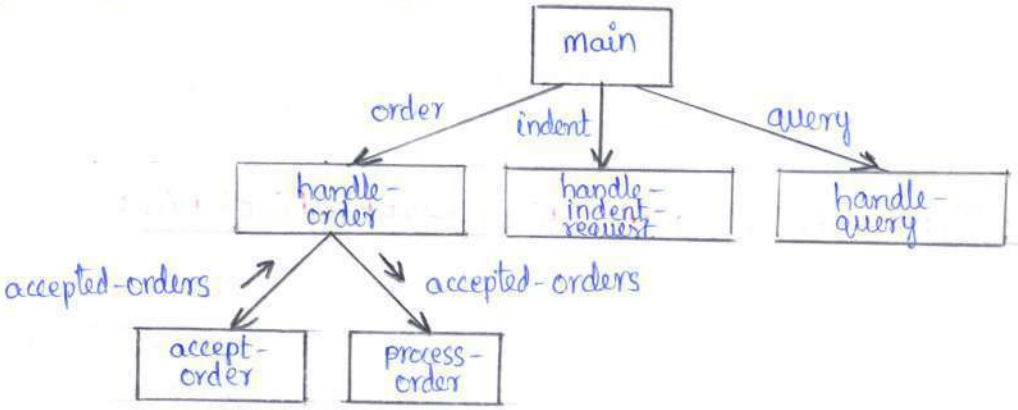


Fig: Structure Chart

The MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

DESIGN REVIEW

After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing and maintenance perspectives, who may or may not be the members of the development team. Normally, the members of the team who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents for the following:

Traceability: The members of the team check whether each bubble of the DFD can be traced to some module in the structure chart and vice versa. They check whether each class functional requirement in

the SRS document can be traced to some bubble in the DFD model and vice-versa.

Correctness : They check whether all the algorithms and data structures of the detailed design are correct.

Maintainability : They check whether the design can be easily maintained in future.

Implementation : They check whether the design can be easily and efficiently implemented.

CHARACTERISTICS OF A GOOD USER INTERFACES

Development of good user interface usually takes significant portion of the total system development effort. For many interactive applications, as much as 50% of the total development effort is spent on developing the user interface part. Unless the user interface is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously. There are few important characteristics of a good user interface.

1. Speed of learning :- The speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorize commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. The speed of learning characteristics of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the s/w.

2. Speed of use :- The speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.

3. Speed of recall :- The speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedure, and intuitive command names.

4. Error prevention :- A good user interface should minimize the scope of committing errors while initiating different commands. The error

rate of an interface can be easily determined by monitoring the errors committed by an average user while using the interface.

5. Attractiveness : An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over the text-based interfaces.

6. Consistency :- The basic purpose of interface consistency is to help users generalize the knowledge of aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reducing the error rate.

7. Feedback :- A good user interface must provide feedback to various user actions. Especially, if any user request takes more than a few second to process, the user should be informed about the state of the processing of his request.

8. Support for multiple skill levels : This is necessary because users with different levels of experience in using an application prefer different types of user interfaces.

9. Error recovery (undo facility) :- While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors and they feel irritated, helpless, and out of control.

10. User guidance and online help :- Users seek guidance and online help when they either forget a command or aware of some features of the s/w. Whenever user needs a guidance or seek help from the system, they should be assisted in overcoming the problem at hand.

User Guidance and Online Help :

Users may seek ~~help~~ help to learn about the operation of the s/w any time while using the s/w. This is provided by the online help system. This help is different from the guidance and error messages

Which are flashed automatically without asking for them. The guidance messages prompt the user with the options he has regarding the next command, and the status of the last command, etc.

- Online help system : A good online help system should keep track of what a user is doing while invoking the help system and provide the output message.
- Guidance Messages : These messages should be carefully designed to prompt the user to the next actions might pursue.
- Error Messages : Error messages are generated by a system either when the user commits some error or when some errors are encountered by the system during processing.

MODE-BASED VS. MODELESS INTERFACE

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the s/w. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the s/w. On the other hand, in a mode-based interface different sets of commands can be invoked depending on the mode in which the system is i.e. the mode at any instant is determined by the sequence of commands already issued by the user.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

TYPES OF USER INTERFACES

User interface can be classified into three categories:

- Command-language-based interfaces
- Menu-based interfaces
- Direct-manipulation interfaces

- Command Language-based Interface : It 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. Command language-based interfaces allow fast interaction with computer and simplify the input of complex commands.
- Menu-Based Interface : An important advantage of a menu-driven interface over a command language-based interface is that the former 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.
- Direct Manipulation Interfaces : Direct manipulation interfaces present the interface to the user in the form of visual models. For this reason, direct manipulation interfaces are sometimes called iconic interfaces. In this type of interfaces, the user issues commands by performing actions on the visual representations of the objects, e.g. pull an icon representing a file into an icon representing a trash box, for deleting the file.

COMPONENT-BASED GUI DEVELOPMENT :

GUI became popular in the 1980s. The main reason why there were very few GUI-based applications prior to the 1980s is that graphics terminals were too expensive. One of the first computers to support GUI-based applications was the Apple Macintosh computer. The current style of user interface development is component-based. It recognizes that every user interface can easily be built from a handful of predefined components such as menus, dialog boxes,

forms, etc. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

1. Window System :- A window system can generate displays through a set of windows. Since a window is the basic entity in such a graphical user interface.

Window :- A window is a rectangular area on the screen. A window can be divided into two parts : client and non-client part. The exiter client are makes up the whole of the window, except for the borders and the scroll bars. The non-client-part of the window determines the look and feel of the window. The window manager is responsible for managing and maintaining the non-client area of a window.

Window management System (WMS) : It is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen.

A WMS consists of two parts :

- A window manager
- A window system

Window manager and window system : Window manager is the component of WMS with which the end-user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.

2. Types of widgets : Different interface programming packages support different widget sets

- Label widget : which displays a label
- Container widget : Other widgets are created as children of container widget.
- Pop-up menu : It appears upon pressing the mouse button.
- Pull-down menu :- These are more permanent and general.
- Dialog boxes :- It remains visible until explicitly dismissed by user.
- Push button : It contains keywords or pictures that describe the action that is triggered when you activate the button.

- Radio buttons : It is used when only one option has to be selected out of many options.
- Combo boxes : It looks like a button until the user interacts with it.

3. An overview of X-Window/Motif : It allows development of portable GUIs. The applications developed using the X-Window system are device-independent, and network independent. The X-window functions are low-level functions written in C language which can be called from application programs.

4. X-Architecture :-

- X-Server :- The X-Server runs on the h/w to which the display and the keyboard are attached.
- X-Protocol :- It defines the format of the requests between client applications and display servers over the network.
- X-Library :- It provides low-level primitives for developing a user interface.
- X-toolkit :- The X-toolkit consists of two parts : the intrinsics and the widgets.

5. Visual Programming : It is the drag and drop style of program development. In this style of user interface development, a number of visual objects representing the GUI components are provided by the programming environment.

6. Size Measurement of a Component-based GUI : LOC is not an appropriate metric to estimate and measure the size of a component-based GUI. This is because the interface is developed by integrating several pre-built components.

USER INTERFACE DESIGN METHODOLOGY

GUI DESIGN METHODOLOGY

The GUI design methodology is based on the seminal work of Frank Ludolph, user interface design methodology consists of the following important steps.

- Examine the use case model of the s/w. Interview, discuss and review the GUI issues with the end-users.
- Task and object modelling
- Metaphor selection
- Interaction design and rough layout
- Detailed presentation and graphics design
- GUI construction
- Usability evaluation

The starting point for GUI design is the use case model. It captures the important tasks the users need to perform using the s/w. As far as possible, a user interface should be developed using one or more metaphors. Metaphors help in interface development at lower effort and reduced costs for training the users. Overtime, people have developed efficient methods of dealing with some commonly occurring situations. These solutions are the themes of the metaphors. Metaphors can also be based on physical objects such as a visitor's book, a catalogue, a pen, a brush, a pair of scissors, etc. A solution based on metaphors is easily understood by the users; it reduces learning time and training costs. Some commonly used metaphors are the following:

- While board
- Editor's work bench
- Office cabinet
- Visitor's book
- Shipping cart
- While page
- Post box
- Desktop
- Yellow page
- Bulletin board

— @ End of UNIT-III @ —

UNIT-IV

Coding and Testing: Coding standards and guidelines, code review, software documentation, Testing, Black Box Testing, White Box Testing, debugging, integration testing, Program Analysis Tools, system testing, performance testing, Testing Object Oriented programs.

CODING STANDARDS & GUIDELINES

Normally, good s/w development organizations require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most s/w development organizations formulate their own coding standards that suits them most, and require their engineers to follow these standards rigorously due to the following reasons.

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It provides sound understanding of the code
- It encourages good programming practices.

Good s/w development organizations usually develop their own coding standards and guidelines depending on what best suits their needs and the types of products they develop.

Reprentative coding standards

The following are some representative coding standards.

1. Rules for limiting the use of globals.
2. Contents of the headers preceding codes for different modules:
   - Name of the module
   - Date on which the module was created
   - Author's name
   - Modification history
   - Synopsis of the module
   - Different functions supported
   - Global variables accessed/modified by the module

3. Naming conventions for global variables, local variables and constant identifiers.
4. Error return conventions and exception handling mechanisms.

Representative Coding Guidelines

The following are some representative coding guiddelines recommended by many s/w development organizations.

1. Do not use a coding style that is too clever or too difficult to understand.
2. Do not use an identifier for multiple purposes.
3. Avoid obscure side effects.
4. The length of any function should not exceed 10 source lines.
5. The code should be well documented.
6. Do not use goto statements. (Use of goto statement makes a program unstructured and very difficult to understand)

CODE REVIEW :- 1. Code Walk-Throughs:

Code Review for a module is an informal code analysis technique. In this technique, after a module has been coded, it is successfully compiled and all syntax errors are eliminated. Some members of the development team are given the code a few days before the walk-through meeting to read and understand the code. Each member selects some test cases and simulates execution of the code by hand. The main objectives of the walk-through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk-through meeting where the coder of the module is also present. Some of the guidelines are as follows:

- The team performing the code walk-through should not be either too big or too small. Ideally, it should consist of three to seven members.

- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among the engineers that they are being evaluated in the code-walk-through meeting, managers should not attend the walk-through meetings.

2. Code Inspection: Code Inspection discovers some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code-walkthroughs. Good s/w development companies collect statistics to identify the type of errors most frequently committed by their engineers. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

The following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables
- Jumps into loops
- Nonterminating loops
- Incompatible assignments
- Array indices out of bounds
- Improper storage allocation and deallocation
- Mismatches between actual and formal parameters in procedure calls
- Use of incorrect logical operators or incorrect precedence among operators
- Improper modification of loop variables
- Comparison of equality of floating point values, etc.

3. Clean Room Testing : Clean Room Testing was pioneered by IBM. This type of testing relies heavily on walk-throughs, inspections, and formal verification. The programmers are not allowed to test any of their code by executing the code, other than doing some syntax testing using a compiler.

This technique reportedly produces documentation and code that is more reliable and maintainable than that obtained from other development methods which rely heavily on code execution-based testing. The main problem with this approach is that testing effort is increased because walk-throughs, inspections, and verifications are time consuming.

SOFTWARE DOCUMENTATION

Good documents are very useful and serve the following purposes:

- Good documents enhance understandability and maintainability of a s/w product. They reduce the effort and time required for maintenance.
- Good documents help the users in effectively exploring the system.
- Good documents help in effectively overcoming the manpower turnover problem.
- Good documents helps the manager in effectively tracking the progress of the project.

Different types of s/w documents can be broadly classified into the following:

- Internal documentation
- External documentation (Supporting documents)

Internal documentation comprises the code comprehension features provided as part of the source code itself. Internal docu-

mentation is provided through appropriate module headers and comments embedded in the source code. For example, the following style of code commenting does not in any way help in understanding the code.

```
a = 10; /* a made 10 */
```

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

An important feature of good documentation is consistency. Inconsistencies in documents create confusion in understanding the product. Also, all the documents for a product should be up-to-date, they create inconsistency and lead to confusion.

TESTING

The aim of the testing process is to identify all defects existing in a s/w product. However, for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the s/w is error free. This is because of the fact that the input data domain of most s/w products is very large.

What is Testing? Testing a program consists of subjecting the program to a set of test inputs and observing if the program behaves as expected. If program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction. The following are some commonly used terms associated with testing:

- A failure is a manifestation of an error (or defect or bug). But the mere presence of an error may not necessarily lead to a failure.
- A test case is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input,

and O is the expected output of the system.

- A test suite is the set of all test case with which a given s/w product is to be tested.

Verification vs. Validation: Verification is the process of determining whether the output of one phase of s/w development conforms to that of its previous phase, whereas validation is the process of determing whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

Design of Test Cases: Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical s/w systems is either extremely large or infinite. Therefore, we must design an optimal test suite that is of reasonable size and can uncover as many errors existing in the system as possible. Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

```
If (x > y) max = x ;
else max = x ;
```

For the above code segment, the test suite {(x=3, y=2); (x=2, y=3)} can detect the error, whereas a larger test suite {(x=3, y=2); (x=4, y=3); (x=5; y=1)} does not detect the error. Therefore, a systematic approach should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors. There are essentially two main approaches to systematically designing test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only

only the functional specification of the s/w, i.e. without any knowledge of the internal structure of the s/w. For this reason, black-box testing is known as "functional testing". On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of s/w, and therefore the white-box testing is called "structural testing".

Unit Testing

Unit testing is undertaken when a module has been coded and successfully reviewed.

Driver and Stub modules: In order to test a single module, we need a complete environment to provide all that is necessary for execution of the module. That is, besides the module under test, we will need the following in order to be able to test the module:

- The procedures belonging to other modules that the module under test calls.
- Nonlocal data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment are usually not available until they, too, have been unit tested; stubs and drivers are designed to provide the complete environment for a module. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior.

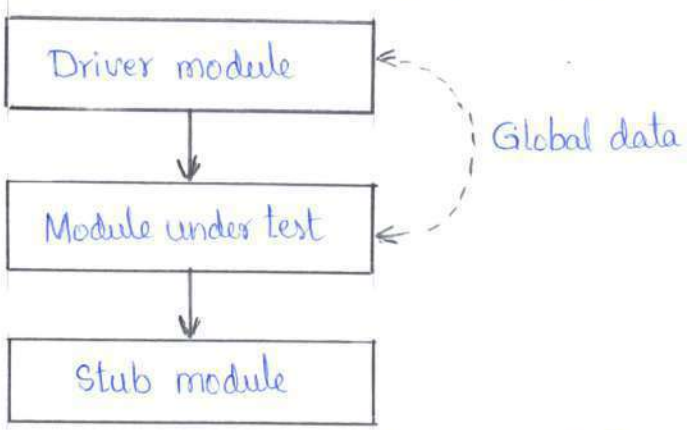


Fig: Unit testing with the help of driver and stub modules.

For example, a stub procedure may produce the expected behaviour using a simple table look-up mechanism. A driver module would 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.

BLACK-BOX TESTING

In Black Box, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black-box text cases.

- Equivalence class partitioning
- Boundary value analysis

Equivalence Class Partitioning In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behaviour of the program is similar to every input data belonging to the same equivalence class. Equivalence classes for a s/w can be designed by examining both the input and the output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another for invalid input values should be defined.

Example: For a s/w that computes the square root of an input integer which can assume values between 0 and 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range between 0 to 5000, and the integers larger than 5000. Therefore, the test cases must include representative values from

each of the three equivalence classes and a possible test set can therefore be: {-5, 500, 6000}.

Boundary Value Analysis:

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= instead of <. BVA leads to selection of test cases at the boundaries of different equivalence classes.

Example: For a function that computes the square root of integer values in the range between 0 and 5000, the test cases must include the following values: {0, -1, 5000, 5001}.

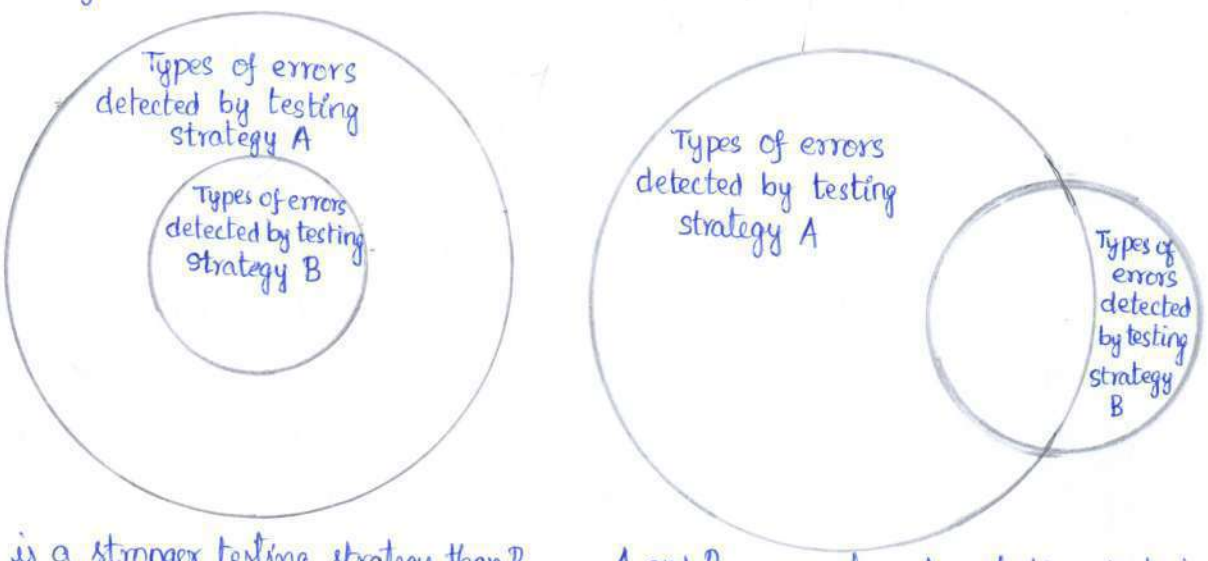
Summary of the Black-Box Test Suite Design:

1. Examine the input and output values of the program.
2. Identify the equivalence classes.
3. Pick the test cases corresponding to equivalence class testing and boundary value analysis.

WHITE-BOX TESTING

There are several white-box testing strategies. Each testing strategy is based on some heuristic. One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy (say B) are also detected by the second testing strategy (say A), and the second 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, they are then called complementary. In the practical design of test cases, if a stronger testing is performed

then the weaker testing need not be carried out. But the test suite should be enriched by using all the complementary testing strategies.



A is a stronger testing strategy than B A and B are complementary testing strategies

Fig: Stronger and Complementary testing strategies

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 we execute a statement, we have no way of determining if an error exists in that statement. Unless a statement is executed, we cannot observe whether it causes failure due to some illegal memory access, wrong result computation, etc.

Example: Consider the following Euclid's GCD Computation algorithm:

```
int compute_gcd(x, y)
    int x, y;
{
    1  while (x != y) {
    2      if (x > y) then
    3          x = x - y;
    4      else y = y - x;
    5  }
    6  return x;
}
```

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed atleast once.

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

3. Condition Coverage : In this structural testing, test cases are designed to make each component of a composite conditional expression assume both true and false values. Condition testing is a stronger testing strategy than branch testing and a branch testing is a stronger testing strategy than the statement coverage-based testing. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

4. Path Coverage : This 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. Therefore, in order to understand the path-coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control Flow Graph (CFG) : It 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, we need to first number all the statement of a program. The different numbered statements serve as the nodes of the control flow graph. 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.

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;
```

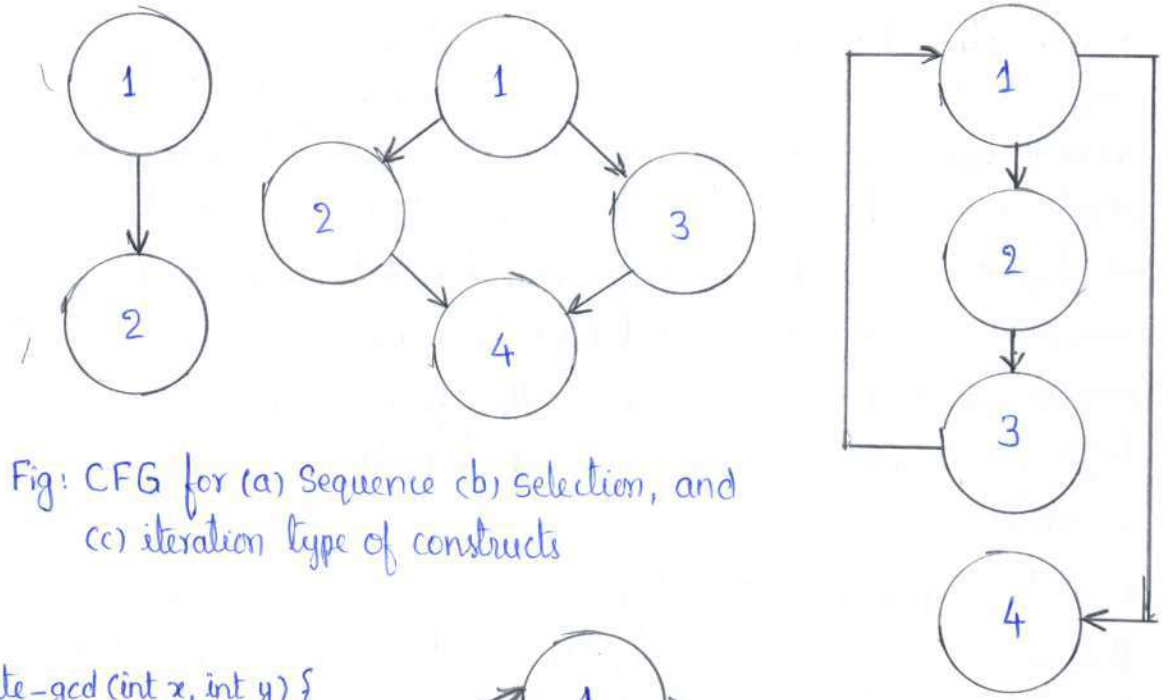
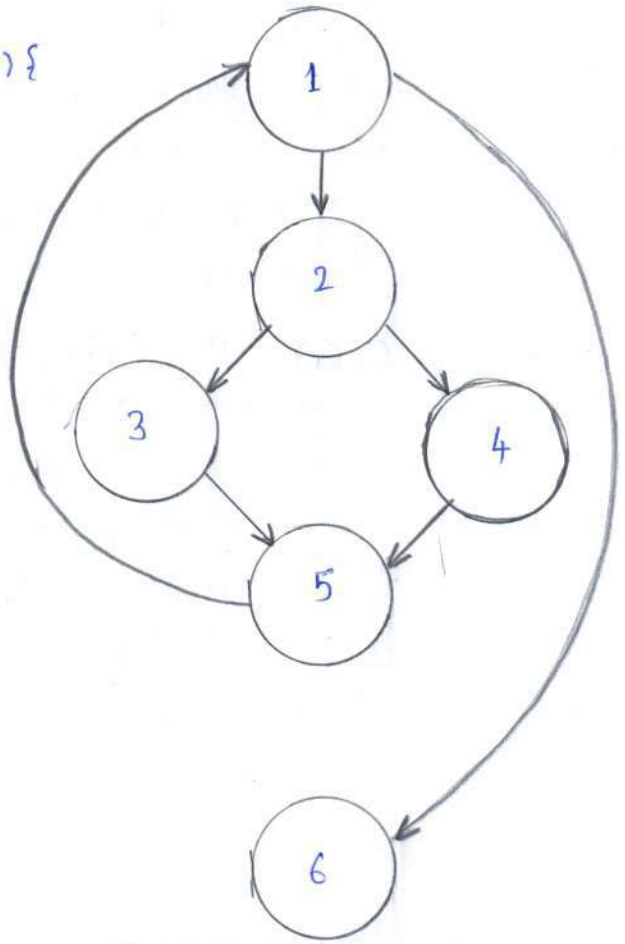


Fig: CFG for (a) Sequence (b) selection, and (c) iteration type of constructs

```
int compute_gcd (int x, int y) {
1   while (x != y) {
2       if (x > y) then
3           x = x - y;
4       else y = y - x;
5   }
6   return x;
}
```

(a) Example program



(b) Control Flow Graph

Fig: Control flow diagram of the program

5. McCabe's Cyclomatic Complexity Metric : It is also called as the structural complexity of the program. It defines an upper bound on the number of independent paths in a program. There are three methods to compute the cyclomatic complexity.

Method 1 : Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as

$$V(G) = E - N + 2$$

Where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example E = 7 and N = 6. Therefore, the cyclomatic complexity, V(G) = 7 - 6 + 2 = 3.

Method 2 : An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

$$V(G) = \text{Total number of bounded areas} + 1$$

Method 3 : The cyclomatic of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N + 1.

6. Data Flow - Based Testing : This testing method select the test paths of a program according to the locations of the definitions and uses of the different variables in a program.

For a statement numbered S, let

DEF(S) = {X | statement S contains a definition of X}, and
USES(S) = {X | statement S contains a use of X}

For the statement S: a = b + c; DEF(S) = {a}. USES(S) = {b, c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.

7. Mutation Testing : In mutation testing, the S/W is first tested by using an initial test suite built up from different white-box testing

strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a mutated program and the change effected is called 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. The process of generation and killing of mutant can be automated by predefining a set of primitive changes that can be applied to the program.

Since mutation testing generates large number of mutants and requires to check each mutant with the full test suite, it is not suitable for manual testing. Mutation test is used in conjunction some testing tool which would run all the test cases automatically.

DEBUGGING

Once errors are identified, it is necessary to first locate the precise program statements responsible for the errors and then to fix them.

Debugging Approaches: The following are some of the approaches popularly adopted by programmers for debugging.

Brute Force 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 as source code debugger), because the 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 is 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 cause. A related technique of identification of the error from the error symptom is the s/w fault tree analysis.

Program Slicing: This technique is equivalent to backtracking. However, the search space is reduced by defining by slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding the statement that can influence the value of that variable.

Debugging Guidelines: Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging.

- Many a times, debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even for simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is that they do not attempt to fix the error but only its symptoms.
- One must be beware of the possibility that any one error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

INTEGRATION TESTING

The primary obtective of integration testing is to test the module interfaces in order to ensure that there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integratein plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested.

An important factor that guides the integration plan is the module dependency graph. Thus by examining structure ~~plan~~ chart the integration plan can be developed. Any one of the following approaches can be used to develop the test plan.

- Big-bang approach
- Top-down approach
- Bottom-up approach
- Mixed approach.

Big-bang integration testing: It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big-bang integration testing are very expensive to fix.

Top-down integration testing: Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top level 'skeleton' has been tested, the immediate subroutines of the 'skeleton' are combine with it and tested.

Top-down integration testing approach requires the use of program stubs to simulate the effect or lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many a times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several lower-level functions such as I/O.

Bottom-up integration testing: In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only the test-drivers are required. A dis-advantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

Mixed integration Testing: A mixed integration testing (also called sandwiched) follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after top level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are

ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when module become available. Therefore, this is one of the most commonly used integration testing approaches.

PROGRAM ANALYSIS TOOLS

A program analysis tools means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc. We can classify these into two broad categories of program analysis tools:

- Static Analysis tools
- Dynamic Analysis tools

Static Program Analysis Tools: Static analysis tool is also a program analysis tool. It assesses and computes various characteristics of a s/w product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

- Whether the coding standards have been adhered to?
- Certain programming errors such as uninialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walkthroughs and code inspections might be considered as static analysis methods. But, the term static program analysis is

used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

Dynamic Analysis tools: Dynamic program analysis techniques require the program to be executed and its actual behaviour recorded. A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces) The instrumented code when executed allows to record the behaviour of the s/w for different test cases. After the s/w has been tested with its full test suite and its behaviour recorded the dynamic analysis tool carries out a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete test suite for the program.

Normally, the dynamic analysis results are reported in the form of a histogram or a pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily and provides evidence that thorough testing has been done. The dynamic analysis results are extent of testing performed in white-box mode. If the testing coverage is not satisfactory more test cases can be designed and added to the test suite. Further, dynamic analysis results can help to eliminate redundant test cases from the test suite.

SYSTEM TESTING

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

- Alpha Testing: Alpha testing refers to the system testing carried out by the test team within the developing organization.
- Beta Testing: Beta testing is the system testing performed by a select group of friendly customers.
- Acceptance Testing: Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the SRS document. Broadly, these tests can be classified into functionality and performance tests. The functionality test tests the functionality of the s/w to check whether it satisfies the functional requirements as documented in the SRS document. The performance test tests the conformance of the system with the nonfunctional requirements of the system.

PERFORMANCE TESTING

Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document. There are several types of performance testing. The types of performance testing to be carried out on a system depends on the different non-functional requirements of the

system documented in the SRS document. All performance tests can be considered as black-box tests.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing
- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

REGRESSION TESTING

This type of testing required when the system being tested in an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

The Regression does not belong to either unit test, integration test, or system testing, it is a separate dimension to these three forms of testing.

TESTING OBJECT ORIENTED PROGRAMS

Testing is a continuous activity during s/w development. In object-oriented systems, testing encompasses three levels, namely unit testing, subsystem testing, and system testing.

Unit testing:

- In unit testing the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free.
- Unit testing is the responsibility of the application engineer who implements the structure.

Subsystem testing:

- This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside.
- Subsystem tests can be used as regression tests for each newly released version of the subsystem.

System Testing:

- System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new.

Object Oriented Testing Techniques:

Grey Box Testing: The different types of test cases that can be designed for testing objected-oriented programs are called grey box test cases. Some of the important types of grey box testing are:

- State model based testing: This encompasses state coverage, state transition coverage, and state transition path coverage.

- Use case based testing: Each scenario in each use case is tested.
- Class diagram based testing: Each class, derived class, associations, and aggregations are tested.
- Sequence diagram based testing: The methods in the messages in the sequence diagram are tested.

Techniques for subsystem Testing:

The two main approaches of subsystem testing are:

- Thread based testing: All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- Use based testing: The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

Categories of system testing:

- Alpha testing: This is carried out by the testing team within the organization that develops s/w.
- Beta testing: This test carried out by selected group of co-operating customers.
- Acceptance testing: This is carried out by the customer before accepting the deliverables.

Black-box testing: Testing that verifies the item being tested when given the appropriate input provides the expected results.

Boundary value testing: Testing of unusual or extreme situations that an item should be able to handle.

Class testing: The act of ensuring that a class and its instances (objects) perform as defined.

Component testing: The act of validating that a component

Works as defined.

Inheritance - regression testing : The act of running the test cases of the super classes, both direct and indirect, on a given subclass.

Integration testing : Testing to verify several portions of s/w work together.

Model review : An inspection ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model.

Path testing : The act of ensuring that all logic paths within your code are exercised atleast once.

Regression testing : The acts of ensuring that previously tested behaviours still work as expected after changes have been made to an application.

Stress testing : The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on.

Technical review : A quality assurance technique in which the design of an application is examined critically by a group of peers. A review typically focuses, on accuracy, quality, usability and completeness. The process is often referred to as a walkthrough, an inspection, or a peer review.

User Interface testing : The testing of the user interface to ensure that it follows accepted user interface standards and meets the requirements defined for it. Often referred to as graphical user interface testing.

White-box testing: Testing to verify that specific lines of code work as defined. Also this testing methodology is known as clear-box testing.

END OF UNIT-4

UNIT-5: SOFTWARE QUALITY, RELIABILITY AND OTHER ISSUES

S/W reliability, Statistical testing, S/W quality and management, ISO-9000, SEI Capability maturity model (CMM), Personal S/W process (PSP), Six sigma, S/W quality metrics, CASE and its scope, CASE environment, CASE support in S/W life cycle, Characteristics of s/w maintenance, S/W reverse engineering, S/W maintenance process model, Estimation maintenance cost, Basic issues in any reuse program, Reuse approach, Reuse at organization level.

SOFTWARE RELIABILITY

Reliability of S/W product essentially denotes its trustworthiness or dependability. Alternatively, reliability of S/W product can also be defined as the probability of the product working correctly over a given period of time. It has been experimentally observed by analyzing the behaviour of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves, due to the correction of a single error, depends on how frequently the corresponding instruction is executed.

Thus, the reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If we select input data to the system such that only the correctly implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high.

The s/w reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a s/w product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

1. Hardware Vs Software Reliability: Reliability behaviour for h/w and s/w is very different. For example, h/w failures are inherently different from s/w failures. Most h/w failures are due to component wear and tear. A logic gate can get stuck at a 1 or 0, or a resistor might have short-circuited. To fix h/w faults, one has to either replace or repair failed part. On the other hand, a s/w product would continue to fail until the error is tracked down either the design or the code is changed. When a s/w failure is repaired, the reliability may increase or decrease.

The changes in failure rate over the product life time for a typical h/w product and a s/w product are sketched below:

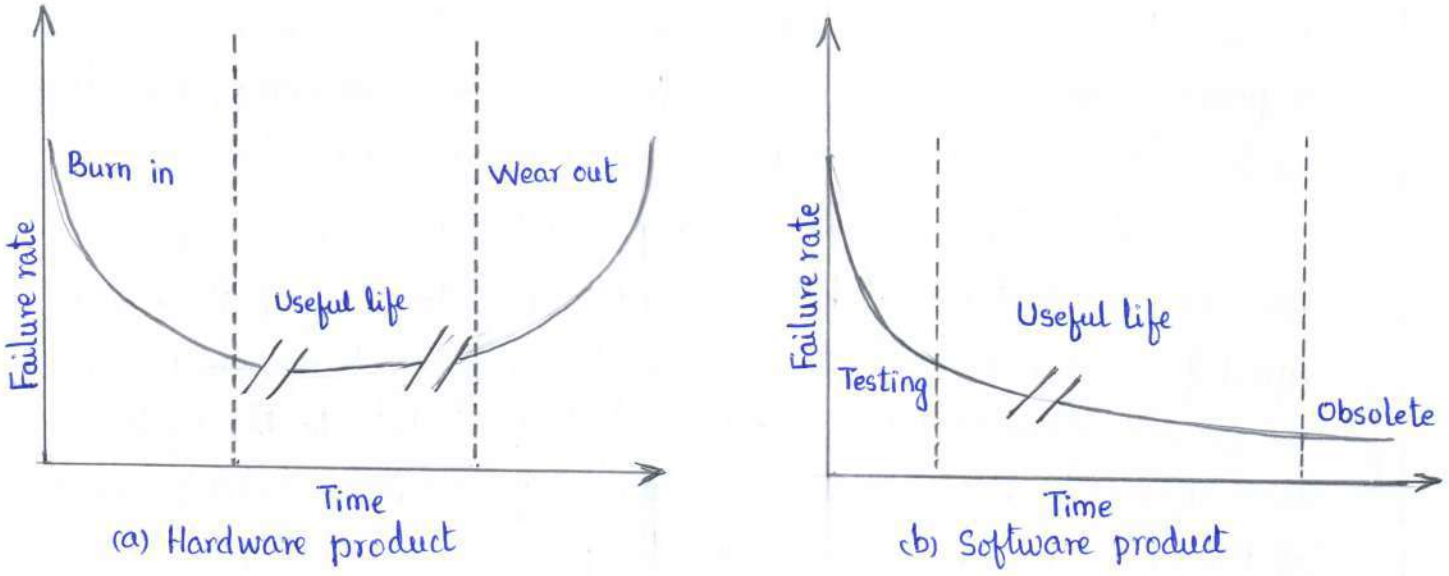


Fig: Change in failure rate of a product

Observe that the failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time the components wear out, and the failure rate increases. This gives the plot of h/w reliability over time its characteristics bath tub shape. On the other hand, for s/w the failure rate is at its highest during integration and testing phase. As the system is tested more and more errors are identified and moved resulting in a reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the s/w becomes obsolete, no more error correction occurs and the failure rate remains unchanged.

2. Reliability Metrics:

The reliability requirements for different categories of s/w products may be different. For this reason, it is necessary that the level of reliability required for a s/w product should be specified in the SRS document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a s/w product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability that a system has. However, in practice, it is very difficult to formulate a precise reliability measurement technique. There are six reliability metrics which can be used to quantify the reliability of s/w products.

1. Rate of Occurrence of Failure (ROCOF): ROCOF measures the frequency of occurrence of unexpected behavior/failures. The ROCOF measure of a s/w product can be obtained by obtaining the behaviour of a s/w product in operation over a specified time interval and then calculating the total number of failures during the interval.

2. 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 $t_1, t_2, \ldots t_n$. Then, MTTF can be calculated as $\sum_{i=1}^{n} \frac{t_{i+1} - t_i}{(n-1)}$.

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

4. Mean Time Between Failures (MTBF) : We can combine the MTTF and MTTR metric to get the MTBF metric : MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected to occur only after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF.

5. Probability of Failure on Demand (POFOD) :- POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

6. Availability : Availability of a system is a measure of how likely will the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time of a system when a failure occurs. This metric is important for systems such as telecommunication systems and operation systems, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

3. Reliability Growth Modelling : A reliability growth model is a mathematical model of how s/w reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level. Although several different

reliability growth models have been proposed, the below are two simple reliability growth models.

Jelinski and Moranda Model: The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since we already know that corrections of different errors contribute differently to reliability growth.

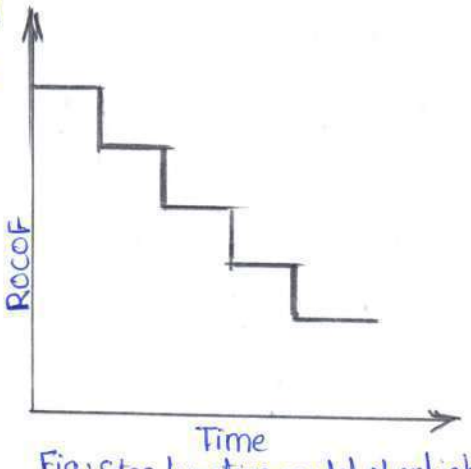


Fig: Step function model of reliability growth

Littlewood and Verall's model: This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement is reliability per repair decreases.

There are more complex reliability growth models, which give greater accurate approximation to the reliability growth. However, these models are beyond the scope of this text.

STATISTICAL TESTING

Statistical testing is a process whose objective is to determine the reliability of the product rather than discovering errors. The test cases designed for statistical testing have an entirely different objective from that of conventional testing. To carry out statistical testing, need to define first the operation profile of the product.

Operation Profile: Different categories of users may use a s/w for very different purposes. Formally, we can define the operation profile of a s/w as the probability distribution of the input of an average user. If we divide the input into a number of classes $\{c_i\}$, the probability value of

a class represents the probability of an average user selecting his next input from this class. Thus, the operation profile assigns a probability value $\{P_i\}$ to each input class $\{c_i\}$.

How to define the operation profile for a product? : We need to divide the input data into a number of input classes. Then need to assign a probability value to each input class; this probability value would signify probability for an input value from that class to be selected. The operation profile of a s/w product can be determined by observing and analyzing the usage pattern of a number of users.

Steps in statistical Testing : The first step is to determine the operation profile of the s/w. The next step is to generate a set of test data corresponding to the determined operation profile. The third step is to apply the test cases to the s/w and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.

For accurate results, statistical testing requires some fundamental assumptions to be satisfied. It requires a statistically significant number of test cases to be used. It further requires that a small percentage of test inputs that are likely to cause system failure be included.

Pros and Cons of Statistical Testing : It allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users find more reliable. Also, the reliability estimation arrived at by using statistical testing is more accurate compared to other methods. However, it is not easy to do statistical testing properly. There is no simple and repeatable way of defining operation profiles. Also, the number of test cases with which the system is to be tested should be statistically significant.

SOFTWARE QUALITY & MANAGEMENT

A quality management system is the principal methodology used by organizations to ensure that the products they develop have the desired quality. A quality system consists of the following:

- **Managerial structure and individual responsibilities:** A quality system is actually the responsibility of the organization as a whole. However, many organizations have a separate quality department to perform several quality system activities. The quality system of an organization should have the support of the top management.
- **Quality System Activities:** The quality system activities encompass the following:
  - Auditing of the projects
  - Review of the quality system
  - Development of standards, procedures, and guidelines, etc.
  - Production of reports for the top management summarizing the effectiveness of the quality system in the organization.

A good quality system must be well documented. Without a properly documented quality system, the application of quality controls and procedures become ad hoc, resulting in large variations in the quality of the products delivered. Also an undocumented quality system sends clear messages to the staff about the attitude of the organization towards quality assurance. International Standards such as ISO 9000 provide guidance on how to organize a quality system.

Evolution of Quality Systems Quality systems have rapidly evolved over the last five decades. Quality systems of organizations have undergone through four stages of evolution. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control

aims at correcting the causes of errors and not just rejecting the defective products. The next breakthrough in quality systems, was the development of quality assurance principles.

The modern quality paradigm includes certain guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. It goes beyond documenting processes with a view to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization.

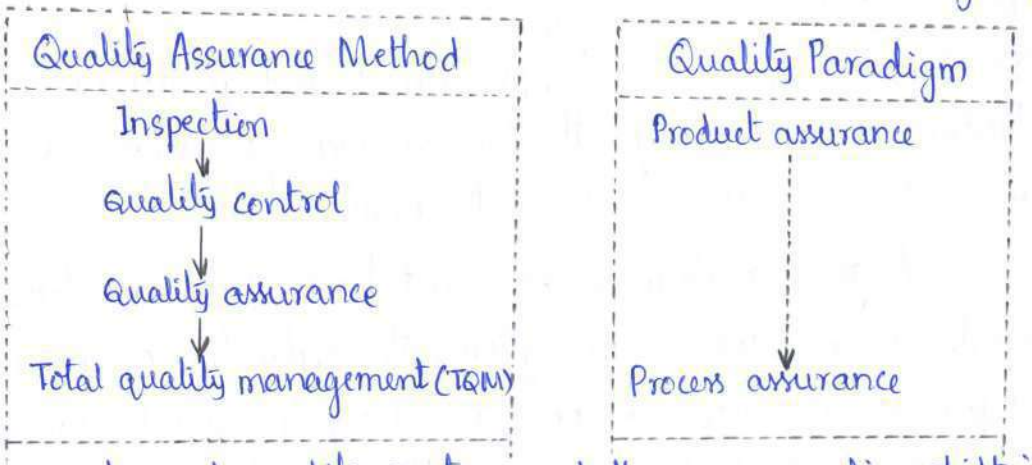


Fig: Evolution of quality system and the corresponding shift in the quality paradigm.

ISO 9000

ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987.

What is ISO 9000 Certification? ISO 9000 certification serves as a reference for contract between independent parties. The ISO9000 standard

specifies the guidelines for maintaining a quality system. The ISO 9000 standard mainly addresses operational aspects of organizational aspects such as responsibilities, reporting, etc. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned with the product itself.

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003. The types of s/w industries to which the different ISO standards apply are as follows:

ISO 9001: This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most s/w development organizations.

ISO 9002: This standard is applied to those organizations which do not design products but are only involved in production. Example of this category of industries include steel and car manufacturing industries, who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to s/w development organizations.

ISO 9003: This standard is applies to organizations involved only in installation and testing of the products.

ISO 9000 for S/W Industry: ISO 9000 is a generic standard that is applicable to a large gamut of industries, starting from a steel manufacturing industry to a service rendering company. Therefore, many of the clauses of ISO 9000 documents are written using generic technologies and it is very difficult to interpret them in the context of s/w development organizations. There are two primary reasons behind this:

- s/w is intangible and therefore difficult to control. It is difficult to control and manage anything that we cannot see and feel. Therefore, it is easy to accurately determine how much work has been completed and to estimate how much more time will it take.
- During s/w development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product. ISO 9000 standards have many clauses corresponding to raw material control. These are obviously not relevant to s/w development organizations.

Due to such radical differences between s/w and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of s/w industry. Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for s/w industry.

Why Get ISO 9000 Certification?

- Confidence of customers in an organization increases when the organization qualifies for ISO certification. This is especially true in the international market.
- ISO 9000 requires a well-documented s/w production process to be in place. A well-documented s/w production process contributes to repeatable and higher quality of the developed s/w.
- ISO 9000 makes the development process focused, efficient, and cost effective.
- ISO 9000 certification points out the weak points of an organization and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and TQM.

How to get ISO 9000 Certification?

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

Application: Once an organization decides to go for ISO 9000 certification, it applies to a registrar for registration.

Pre-assessment: During this stage, the registrar makes a rough assessment of the organization.

Document review and adequacy of audit: During this stage, the registrar reviews the documents submitted by the organization and makes suggestions for possible improvements.

Compliance audit: During this stage, the registrar checks whether the suggestions made by it during review have been compiled with by the organization or not.

Registration: The registrar awards the ISO 9000 certificate after successful completion of all previous phases.

Continued Surveillance: The registrar continues to monitor the organization, through periodically.

ISO mandates that a certified organization can use the certificate for corporate advertisements but cannot use the certificate for advertising any of its products. This is probably due to the fact that the ISO 9000 certificate is issued for an organization's process and does not apply to any specific product of the organization. An organization using ISO certificate for product advertisements faces the risk of withdrawal of the certificate. In India, ISO 9000 certification is offered by (BIS) Bureau of Indian Standards, STQC (Standardization, Testing, and Quality Control), and IRQS (Indian Register Quality System). IRQS has been accredited by the Dutch council of certifying bodies.

SEI CAPABILITY MATURITY MODEL (SEICMM)

Software Engineering Institute Capability Maturity Model helped organizations to improve the quality of the s/w they develop and therefore adoption of SEICMM model has significant business benefits.

SEI CMM can be used in two ways: capability evaluation and s/w process assessment. Capability evaluation and s/w process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the s/w capability of an organization. The results of capability evaluation indicates the likely contractor performance if the contractor is awarded a work. Therefore, the results of s/w process capability assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

SEI CMM classifies s/w development industries into the following five maturity levels. The different levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality systems starting from scratch.

Level 1 : Initial. A s/w development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed. Since s/w production processes are not defined different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depends on individual efforts and heroics. When engineers leave, the successors have great difficulty in understanding the process followed and the work completed. Since formal project management practices are not followed, under time pressure shortcuts are tried out leading to low quality.

Level 2: Repeatable. At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications.

Level 3: Defined. At this level the process for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles and responsibilities. The process though defined, the process and product qualities are not measured. ISO 9000 aims at achieving this level.

Level 4: Managed. At this level, the focus is on s/w metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. The s/w process and product quality are measured and quantitative quality requirements for the product are met. Various tools like Pareto charts, Fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimizing At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback

from the process measurements and also from application of innovative ideas and technologies. Such an organization identifies the best s/w engineering practices and innovations which may be tools, methods, or processes. These best practices are transferred throughout the organization.

Key process areas (KPA) of a s/w organization:

Except for SEI CMM level 1, each maturity level is characterized by several key process areas that includes the areas an organization should focus to improve its s/w process to the next level. The focus of each level and the corresponding key process areas are shown in the below table:

CMM Level	Focus	Key process areas
1. Initial	Competent people	
2. Repeatable	Project Management	S/w project planning S/w Configuration Mgmt.
3. Defined	Definition of processes	Quantitative process metrics S/w quality management
4. Managed	Product and process quality	Process definition Training program Peer reviews
5. Optimizing	Continuous process improvement	Defect prevention Process change management Technology change mgmt.

Fig: The focus of each SEI CMM level and the corresponding Key process areas

SEI CMM provides a list of key process areas on which to focus to take an organization from one level of maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such that one

stage enhances the capability already built up. For example, it considers that trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.

Applicability of SEI CMM to organizations:

SEI CMM model is perfectly applicable for large organizations. But small organizations typically handle applications such as internet, e-commerce, and are without an established product range, revenue base, and experience on past projects, etc. For such organizations, a CMM-based appraisal is probably excessive. These organizations need to operate more efficiently at the lower levels of maturity. For example, they need to practice effective project management, reviews, configuration management, etc.

PERSONAL SOFTWARE PROCESS (PSP)

Personal S/w Process is a scaled down version of the industrial s/w process. PSP is useful for individual use. PSP recognizes that the process for individual use is different from that necessary for a team. The quality and productivity of an engineer is to a great extent dependent on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating and planning, by showing how to track performance against plans, and provides a defined process which can be tuned by individuals.

Time Measurement: PSP advocates that engineers should rack the

the way they spend time. Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-clock to get an objective picture of the time spent. An engineer should measure the time he spends for designing, writing code, testing, etc.

PSP Planning: Individuals must plan their project. They must estimate the maximum, minimum, and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum, and the average development time. They must record the plan data in a project plan summary.

The PSP is schematically shown in below figure, while carrying out the different phases, they must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve their process, etc.

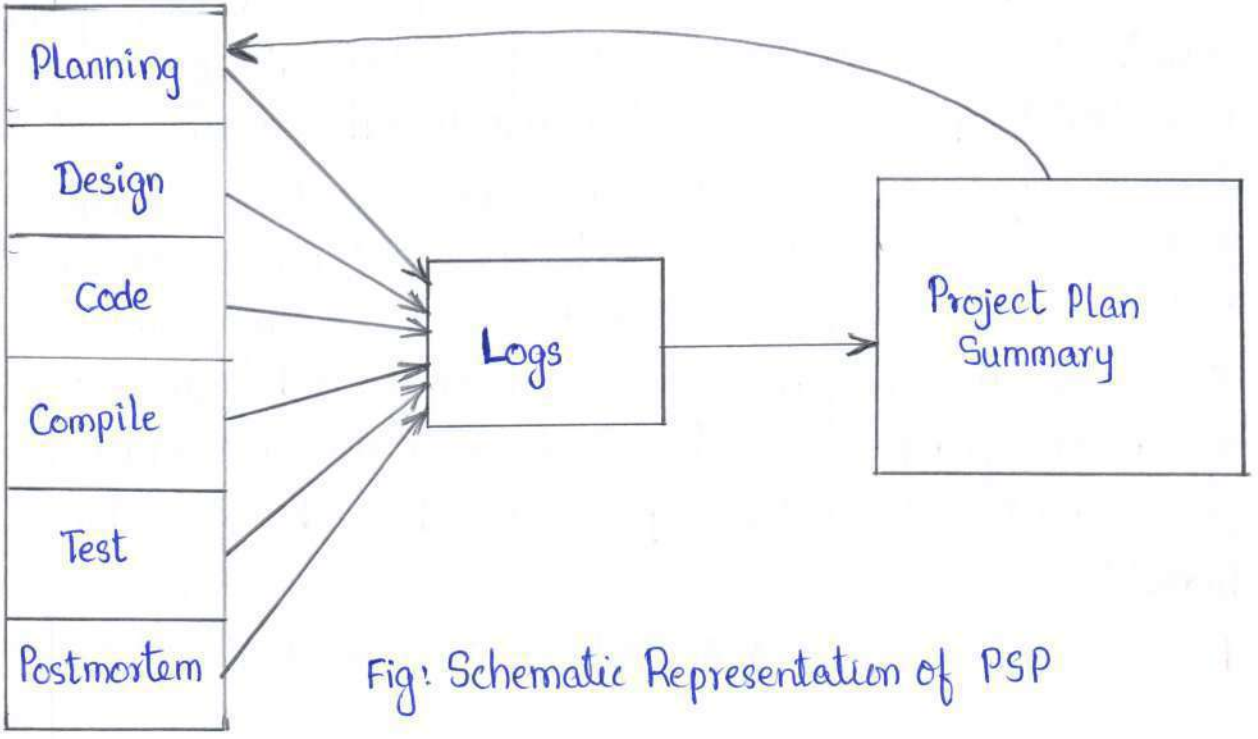


Fig: Schematic Representation of PSP

PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed from gathering and analyzing defect data earlier projects.

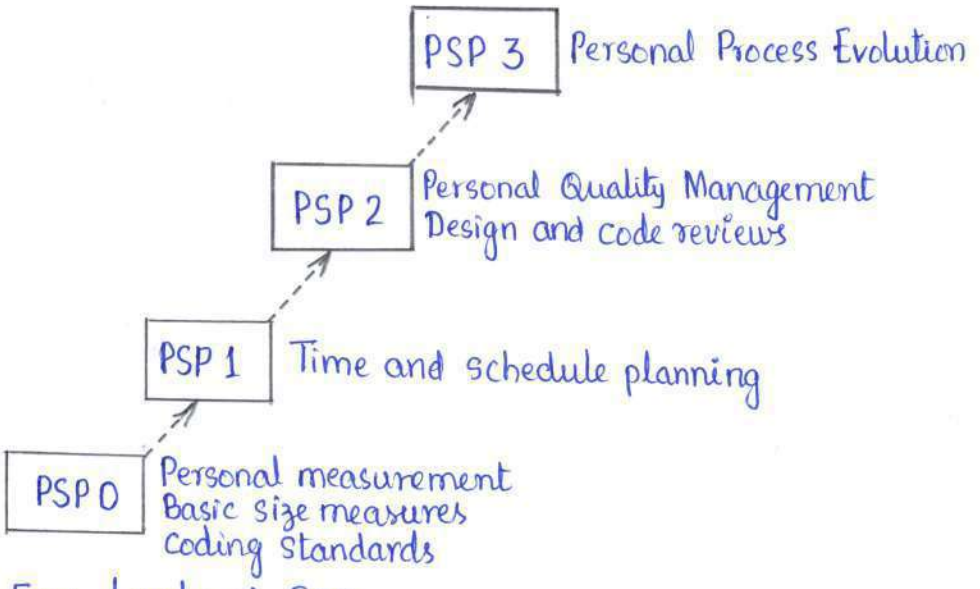


Fig: Levels of PSP

SIX SIGMA

The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost. It can be used to improve every facet of business, from production to human resources, to order entry, to technical support. Six Sigma can be used for any activity that is concerned with cost, timelines, and quality of results. Therefore, it is applicable virtually to every industry.

Six Sigma at many organizations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process - from manufacturing to transactional and product to service.

The statistical representation of Six Sigma describes quantitatively how a process is performing. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. A Six Sigma defect is defined as any system behavior

that is not as per customer specifications. Total number of Six Sigma opportunities is then the total number of chances for a defect. Process sigma can easily be calculated using a Six Sigma calculator.

The fundamental objective of Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two six sigma sub-methodologies: DMAIC and DMADV. The Six Sigma DMAIC process (define, measure, analyze, improve, control) is an improvement system for existing processes falling below specification and looking for incremental improvement. The Six Sigma DMADV (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts.

Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

SOFTWARE QUALITY METRICS

S/w Quality Metrics are a subset of s/w metrics that focus on the quality aspects of the product, process and project. These are more closely associated with process and product metrics than project metrics.

S/w quality metrics can be further divided into three categories :

- Product quality metrics
- In-process quality metrics
- Maintenance quality metrics

1. Product Quality Metrics : This metrics includes the following:
   - Mean Time to Failure
   - Defect Density
   - Customer Problems
   - Customer Satisfaction

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

Defect Density : It measures the defects relative to the s/w size expressed as lines of code or function point, etc. i.e. it measures code quality per unit. This metrics is used in many commercial s/w systems.

Customer Problems : It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the s/w, which includes the non-defect oriented problems together with the defect problems. The problems metric is usually expressed in terms of problems per User-Month (PUM).

PUM = Total problems that customers reported (true defect and non-defect oriented problems) for a time period + Total number of license months of the s/w during the period.

Where, Number of license-months of the software = Number of install license of the s/w X Number of months in the calculation year.

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

Customer Satisfaction: It is often measured by customer survey data through the five point scale-

- Very satisfied
- Satisfied
- Neutral
- Dissatisfied
- Very dissatisfied

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

- Percent of completely satisfied customers
- Percent of satisfied customers
- Percent of dissatisfied customers
- Percent of non-satisfied customers

2. In-Process Quality Metrics: In-process Quality Metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes-

- Defect density during machine testing
- Defect arrival pattern during machine testing
- Phase-based defect removal pattern
- Defect removal effectiveness

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

- Fix backlog and backlog management index
- Fix response time and ~~and~~ fix responsiveness
- Percent delinquent fixes
- Fix quality

CASE AND ITS SCOPE :

CASE stands for Computer Aided S/w Engineering. CASE tools promise reduction in S/w development and maintenance costs. CASE tools help develop better quality products more efficiently.

A CASE tool is a generic term used to denote any form of automated support for S/w Engg. Many CASE tools are now available. Some of these tools assist in phase-related tasks such as specification, structured analysis, design, coding, testing, etc, and others are related to non-phase activities such as project management and configuration management. The primary objectives of deploying CASE tool are :

- To increase productivity
- To produce better quality S/w at lower cost.

CASE ENVIRONMENT

CASE tools are characterized by the stage or stages of s/w development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the s/w. This central repository usually a data dictionary containing the definitions of all composite and elementary data items. Through the central repository, all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilitates the automation of the step-by-step methodologies for s/w development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of s/w development. The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger. The different tools are integrated to the extent that once the compiler detects an error, the editor automatically goes to the statements in error and the error statements are highlighted. Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc. A schematic representation of a CASE environment, standard programming environments such as Turbo C, Visual C++, and more come equipped with a program editor, compiler, debugger, linker, etc. All these tools are integrated. If you click on an error reported by the compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

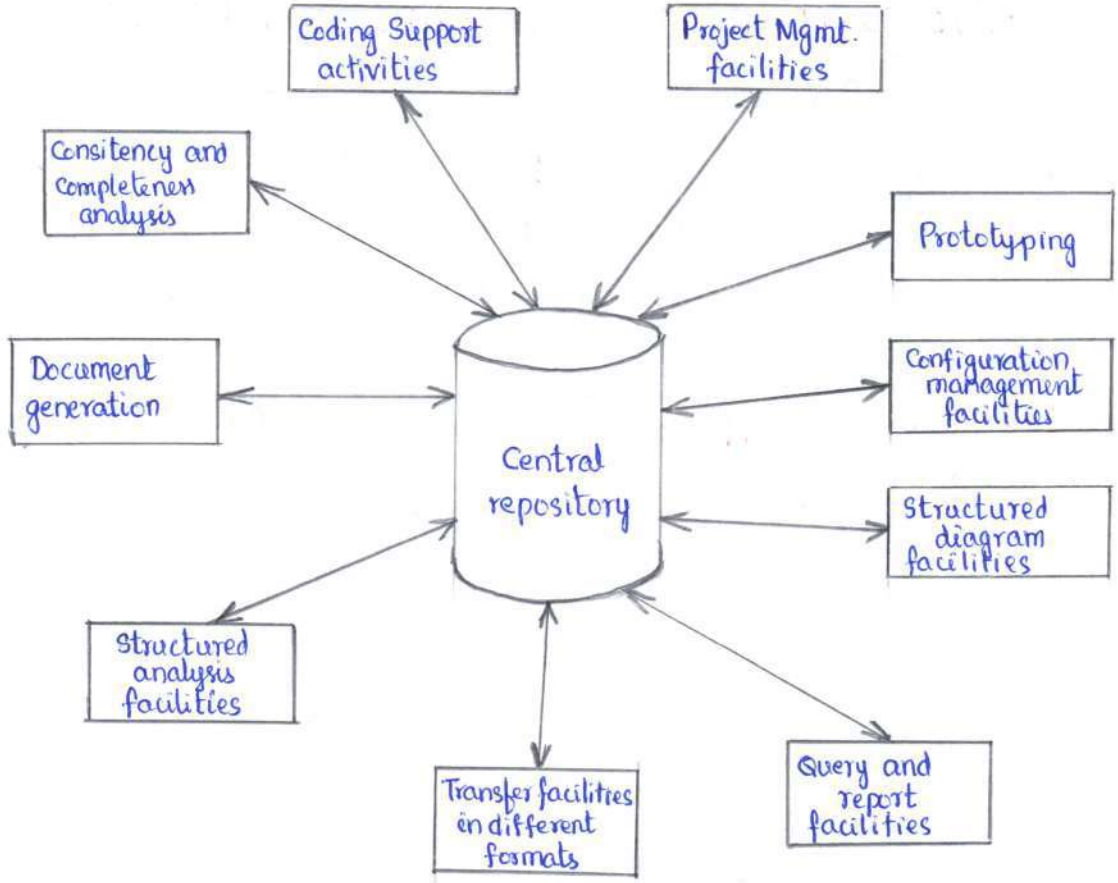


Fig: A CASE environment

Benefits of CASE: Several benefits accrue from the use of a CASE environment or even from the use of isolated CASE tools. Let us examine some of these benefits:

- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases. CASE put the effort reduction between 30% to 40%.
- Use of CASE tools leads to considerable improvements to quality.
- CASE tools help produce high quality and consistent documents.
- CASE tools reduce the drudgery (lazy at work) in a s/w engineer's work.
- CASE tools have led to revolutionary cost savings in s/w maintenance efforts.
- Use of a CASE environment has an impact on the style of working of a company, and makes it conscious of structured and orderly approach.

CASE Support in Software Life Cycle

CASE tools should support a development methodology, help enforce the same, and provide certain amount of consistency checking between different phases. The kind of support that CASE tools usually provide in the s/w development life cycle is elucidated below:

1. **Prototyping Support**: The prototyping CASE tool's requirements are as follows:
   - Define user interaction
   - Define the system control flow
   - Store and retrieve data required by the system
   - Incorporate some processing logic

There are several standalone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype.

A good prototyping tool should support the following features:

- A prototyping CASE tool should support the user to create a GUI using a graphic editor. The user should be allowed to define all data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.
- If possible, it should be able to integrate with the external user-defined modules written in C or in some popular high level programming languages.
- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allow to control the running of the prototype.

- The run-time system of the prototype should support mock up run of the actual system and management of the input and output data.

2. Structured Analysis and Design: Several diagramming structures techniques are used for structured analysis and design. A CASE tool should support one or more of the structured analysis and design techniques. It should support effortlessly making of the analysis and design diagrams. It should also support making of the fairly complex diagrams and preferably through a hierarchy levels. The CASE tool should provide easy navigation through different levels of design and analysis.

3. Code Generation: As far as code generation is concerned, the general expectation from a case tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic support expected from a CASE tool during the code generation phase comprises the following:

- The CASE tool should support generation of module skeletons or templates in one or more popular programming languages. It should be possible to include copyright message, brief description of the module, author name and date of creation in some selectable format.
- The tool should generate records, structures, class definitions, automatically from the contents of the data dictionary in one or more popular programming languages.
- It should generate database tables for relational database management systems.
- The tool should generate code for user interface from prototype definition for X-Windows, and MS-Windows-based applications.

4. Test CASE Generator: The CASE tool for test case generation should have the following features:

- It should support both design and requirement testing.
- It should generate test set reports in ASCII format which can be directly imported into the test plan document.

CHARACTERISTICS OF SOFTWARE MAINTENANCE

S/w maintenance is becoming an important activity of a large number of organizations. This is no surprise, given the rate of h/w obsolescence, the immortality of a s/w product, and the demand of the user community to see the existing s/w products run on newer platforms, run in newer environments, and/or with enhanced features. Also, whenever the support environment of a s/w product changes, the s/w product requires rework to cope with the newer interface. Thus, every s/w product continues to evolve after its development through maintenance efforts.

1. Types of s/w maintenance: The requirement of s/w maintenance arises on account of the three main reasons:

Corrective: Corrective maintenance of a s/w product becomes necessary to rectify the bugs observed while system is in use.

Adaptive: A s/w product might need maintenance when the customers need the product to run on new platforms, on new operating systems or when they need the product to be interfaced with new h/w or s/w.

Perfective: A s/w product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

2. Characteristics of S/W Evolution : Lehman and Belady have studied the characteristics of evolution of several s/w products. They have observed in the form of laws.

Lehman's first law : A s/w product must change continually or become progressively less useful.

Lehman's Second Law : The structure of a program tends to degrade as more and more maintenance is carried on it.

Lehman's Third Law : over a program's lifetime, its rate of development is approximately constant.

3. Special Problems Associated with S/W Maintenance :

- During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.
- Another problem associated with maintenance work is that the majority of s/w products needing maintenance are legacy(aged) products.

SOFTWARE REVERSE ENGINEERING

S/W Reverse Engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. The reverse engineering is becoming important, since legacy s/w products lack proper documentation, and are highly unstructured. Even well-designed products becomes legacy s/w as their structure degrades because of a series of maintenance efforts implemented a period of use.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability structure and understandability, without changing any of its functionalities. A program can be formatted using any of the several available pretty-printer programs which layout the program neatly. Many legacy s/w products are difficult to comprehend with complex control structure and unthoughtful variable names. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by 'case' statements.

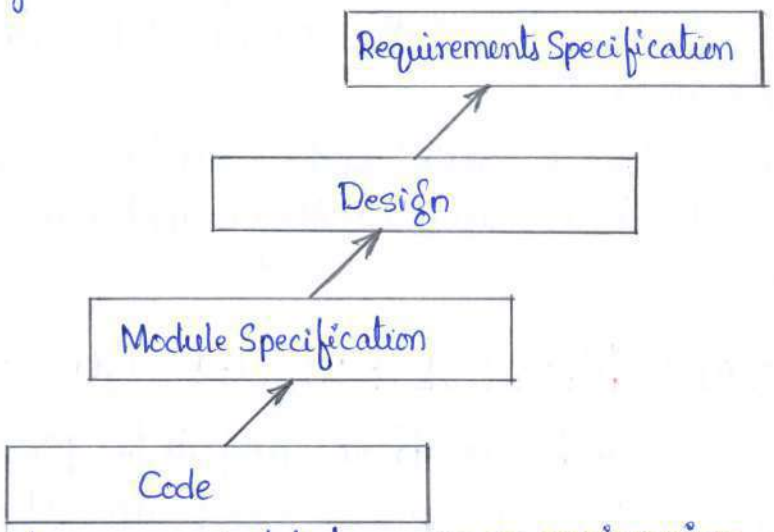


Fig: A process model for reverse engineering

After the cosmetic changes have been carried out on a legacy s/w, the process of extracting the code, design and the requirements specification can begin. Some automatic tools can be used to derive the data flow and the control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

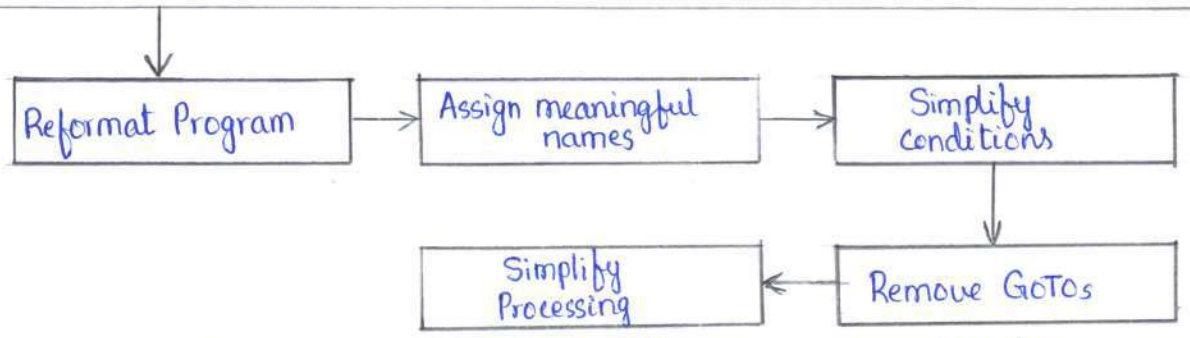


Fig: Cosmetic changes carried out before reverse engineering

SOFTWARE MAINTENANCE PROCESS MODELS

The activities involved in a s/w maintenance project are not unique and depend on several factors such as : (i) the extent of modification to the product required, (ii) the resources available to the maintenance team, (iii) the expected project risks etc. When the changes needed to a s/w product are minor and straightforward the code can be directly modified and the changes approximately, reflected in all the documents.

However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the s/w process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Since the scope for different maintenance projects vary widely, no single maintenance process model can be developed to suit every kind of maintenance project. However, two broad categories of process models can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant document later.

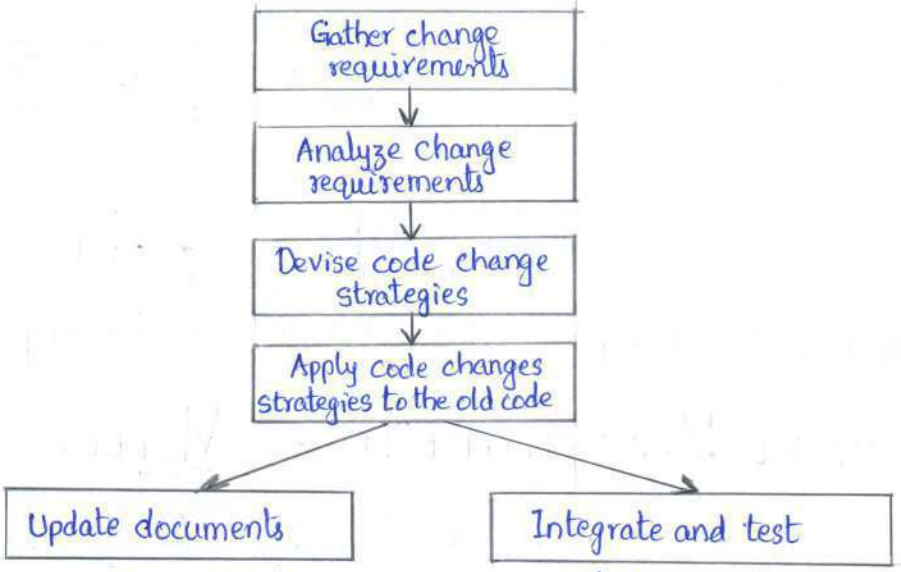


Fig: Maintenance process model-1

In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the engineers as the get a good insight into the working of the old system and also can compare the working of their modified system with the old system

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as s/w reengineering. The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed to extract the module specifications. The module specifications are then analyzed to produce the design. The design is analyzed to produce the original

requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At this point a forward engineering is carried out to produce the new code. At the design, module specification, and coding stages, a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design than what the original product, produces good documentation, and very often results in increased efficiency. However, this approach is costlier than the first one. An empirical study indicates that process 1 is preferred when the amount of re-work is no more than 15%. Besides the amount of rework, several other factors might affect the decision regarding using model 1 over process 2.

- Reengineering might be preferable for products which exhibit a high failure rate.
- Reengineering might also be preferable for legacy problems having poor design and code structure.

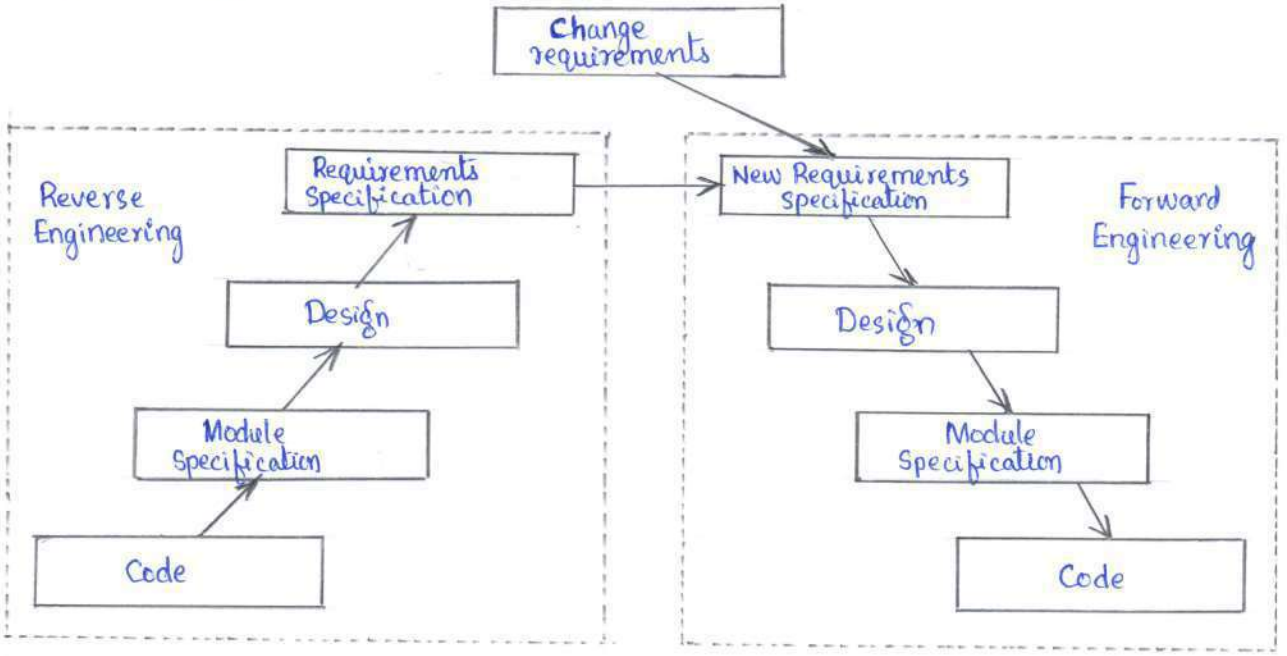


Fig: Maintenance Process Model-2

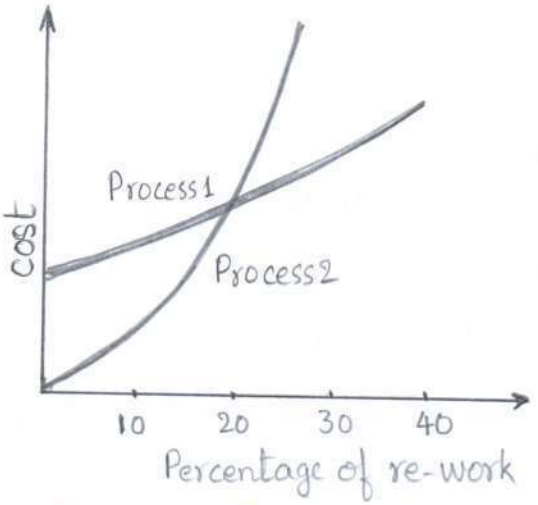


Fig. Empirical estimation of maintenance cost Vs. percentage rework

Estimation of Maintenance Cost

Maintenance efforts constitute about 60% of the total life cycle cost for a typical s/w product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quality called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a s/w product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

Where, $KLOC_{added}$ is the total kilo lines of source code added during maintenance. $KLOC_{deleted}$ is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and the code deleted.

The ACT is multiplied with the total development cost to arrive at the maintenance cost: maintenance cost = ACT x development cost.

Most maintenance cost estimation models, however, give approximate results because they do not take into account several factors such as the experience level of engineers, and familiarity of engineers with the product, h/w requirements, s/w complexity, etc.

BASIC ISSUES IN ANY REUSE PROGRAM

S/w products are expensive. S/w project managers are worried about the high cost of s/w development and are desperately looking for ways to cut development costs. A possible way to reduce the development cost is to reuse parts from the previously developed s/w. In addition to achieving reduced development cost and time, reuse also ensures a high quality of the developed products since the reusable components have already stood the test of reliability.

The following are some of the basic issues that must be clearly understood for starting any reuse program.

1. Component creation: For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important.

2. Compont indexing and storing: Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse. The components need to be stored in a relational database management system (RDBMS) or an object-oriented database system (ODBMS) for efficient access when the number of components becomes large.

3. Component searching: The programmers need to search for right components by matching their requirements with components stored in a database. To be able to search components efficiently, the programmers required a proper method to describe the components that they are looking for.

4. Component understanding: The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding the components should be well documented and should do something simple.

5. Component adaptation : Often, the components may need adaptation before they can be reused, since a selected component not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

6. Repository maintenance : A component repository once created, requires continuous maintenance. New components, as and when created into the repository. The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

A Reuse Approach

A promising approach that is being adopted by many organizations is to introduce a building block approach into the s/w development process. For this, the reusable components need to be identified after every development project is completed. The reusability of the identified components has to be enhanced and these have to be catalogued into a component library.

1. Domain Analysis : The aim of domain analysis is to identify the reusable components for a problem domain.

Reuse domain. A reuse domain is a technically related set of application areas. A body of information is considered to be a problem domain for reuse, if a deep and comprehensive relationship exists among the information items as characterized by patterns of similarity among the development components of the s/w product.

Evolution of reuse domain : The ultimate result of domain analysis is the development of problem-oriented languages. These are also known as application generators. These application generators, once developed from application development standards. The domains slowly develops, we may distinguish the various stages it undergoes :

Stage 1 : There is no clear and consistent set of notations. Obviously, no reusable components are available. All s/w is written from scratch.

Stage 2 : Here, only experience from similar projects is used in a new development effort, this means that there is only knowledge reuse.

Stage 3 : At this stage the domain is ripe for reuse. The set of concepts are stabilized and the notations standardized. Standand solutions to standard problems are available. There is both knowledge and component reuse.

Stage 4 : The domain has been fully explored. The s/w development for domain can be largely automated. Programs are not written in the traditional sense any more. Programs are written using a domain specific language, which is also known as an application generator.

2. Component Classification : Components need to be properly classified in order to develop an effective indexing and storage scheme. At the lowest level, the components are described in several forms : natural language description, logic schema, timing information, etc. The higher the level at which a component is described, the more is the ambiguity. This has motivated the Prieto-Diaz's classification scheme.

Prieto-Diaz's Classification Scheme : Each component is best described using a number of different characteristics or facets. For example, objects can be classified using the following:

- Actions they embody
- Objects they manipulate
- Data structures used
- Systems they are part of, and so forth

Prieto-Diaz's faceted classification scheme requires choosing an n-tuple that best fits a component. Faceted components has advantages over the enumerative classification.

3. Searching : A popular search technique that has proved to be very effective is the one that provides a web interface to the repository. Using such a web interface, one would search for an item using an approximate automated search and using keywords, and then from these results do a browsing using the links provided to lookup the related items.

4. Repository Maintenance : Repository maintenance involves entering new items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. Also, the links relating the

different items may need to be modified to improve the effectiveness of search.

5. Reuse without Modifications: Once the standard solutions emerge, no modifications to the program parts may be necessary. One can plug in the parts directly to develop one's application. Reuse without modification is much more useful than the classical program libraries. These can be supported by compilers through linkage to run-time support routines (application generators).

REUSE AT ORGANIZATION LEVEL

Reusability should be a standard part in all s/w development activities including specification, design, implementation, test, etc. Ideally, there should be a steady flow of reusable components. In practice, things are not so simple.

Extracting reusable components from projects that were completed in the past presents a real difficulty not encountered while extracting a reusable component from an ongoing project - typically, the original developers are no longer available for consultation. Development of new systems can lead to an assortment of products, since reusability ranges from items whose reusability is immediate to those items whose reusability is highly improbable.

Achieving organization-level reuse requires the adoption of the following steps:

- Assessing a product's potential for reuse.
- Refining a product for greater reusability.
- Improving reusability of a product by handling portability problems.

Assessing a product's potential for reuse: Assessment of a component's reuse potential can be obtained from an analysis of a questionnaire circulated among the developers. The questionnaire can be devised to assess a component's reusability. The programmers working in similar application domains can be asked to answer the questionnaire about the product's reusability. Depending on the answers given by the programmers, either the component is

taken up for reuse as it is, or it is modified and refined before it is entered into the reuse repository, or it is ignored. A sample questionnaire to assess a component's reusability as follows:

- Would the component's functionality be required for implementation of systems in the future?
- How common is the component's function within its domain?
- Would there be a duplication of functions within the domain if the component is taken up?
- Is the component h/w dependent?
- Is the design of the component optimized enough?
- If the component is non-reusable, then can it be decomposed to yield some reusable components?
- Can we parametrize a non-reusable component so that it becomes reusable?

Refining products for greater reusability: For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localized using data encapsulation techniques. The following refinements may be carried out:

Name generalization. The names should be general, rather than being directly related to a specific application.

Operation generalization. Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.

Exception generalization. This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.

Handling portability Problems: Programs typically make some assumptions regarding the representation of information in the underlying machine. These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be the same on all machines. Also, programs use some function libraries, which may not be available on all host machines. The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program. Also, all platform-

related calls should be routed through the portability interface. One problem with this solution is the significant overhead incurred, which makes it inapplicable to many real-time systems and applications requiring very fast response.

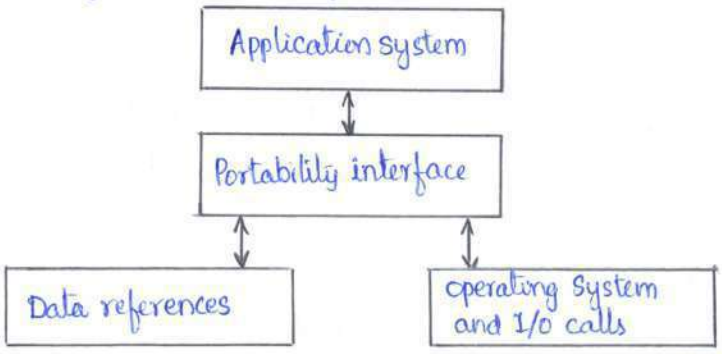


Fig: Improving reusability of a component by using a portability interface

Current State of Reuse: In spite of all the short comings of the state-of-the art reuse techniques, it is the experience of several organizations that most of the factors inhibiting an effective reuse program are non-technical. Some of these factors are the following:

- Need for commitment from the top management
- Adequate documentation to support reuse
- Adequate incentive to reward those who reuse. Both the people contributing new reusable components and those reusing the existing components should be rewarded to start a reuse program and keep it going.
- Providing access to and information about reusable components. Organizations are often hesitant to provide an open access to the reuse repository for the fear of the reuse components finding a way to their competitors.

— * END OF UNIT-5 * —