BSC IInd sem SAD Unit IV - Project Management
by Rayib Mali

responsibilities of a project manager. The responsibilities and activities of a project manager is large and varied. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but we can broadly classify them into two major types of responsibilities of the project manager.

We can broadly classify the different activities of a project manager into project planning and project monitoring and control activities.

We give an overview of these two responsibilities. Later, we discuss them in more detail.

1. Project planning: Project planning involves estimating several characteristics of the project and then planning the project activities based on the estimates made. Project planning is undertaken immediately after the feasibility study phase and before the requirements analysis and specification phase. The initial project plans that are made are revised from time to time as the project progresses and more project data become available.

2. Project monitoring and control activities: The project monitoring and control activities are undertaken once the development activities start. The aim of the project monitoring and control activities is to ensure that the development proceeds as per plan. The plan is changed whenever required to cope up with the situation at hand.

3.1.2 Skills Necessary for Software Project Management

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgement and decision-making capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, and configuration management, etc.; project managers need good communication skills and the ability get work done. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. Nonetheless, the importance of a sound knowledge of the prevalent project management techniques cannot be overemphasized. The objective of the rest of this chapter is to introduce you to the same.

With this brief discussion on the responsibilities and roles of software project managers, in the next section we examine some important issues in project planning.

3.2 PROJECT PLANNING

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

1. Estimation: The following project attributes have to be estimated.

(i) *Cost* How much is it going to cost to develop the software?

(ii) *Duration* How long is it going to take to develop the product?

(iii) *Effort* How much effort would be required to develop the product?

The effectiveness of all other planning activities such as scheduling and staffing are based on the accuracy of these estimations.

2. Scheduling: After the estimations are made, the schedules for manpower and other resources have to be developed.

3. Staffing: Staff organization and staffing plans have to be made.

4. Risk management: Risk identification, analysis, and abatement planning have to be done.

5. Miscellaneous plans: Several other plans such as quality assurance plan, configuration management plan, etc. have to be done.

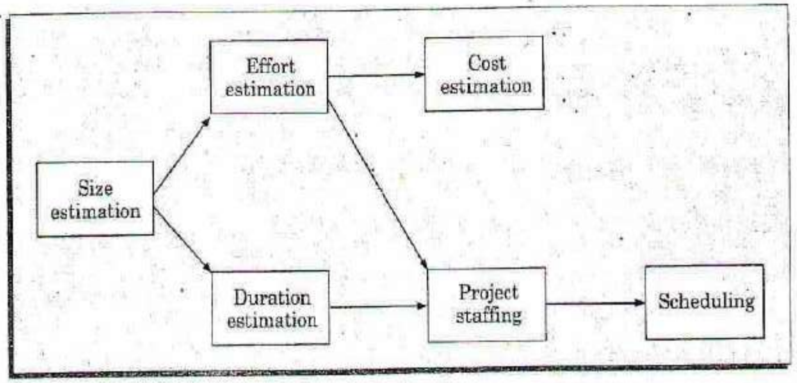

Figure 3.1: Precedence ordering among planning activities.

Figure 3.1 shows the order in which these important planning activities are usually undertaken. Observe that size estimation is the first activity.

Size is the most fundamental parameter based on which all other estimates are made.

Based on the size estimation, the effort required to complete the project and the duration over which the development is to be carried out are estimated. Based on the effort estimation, the cost of the project is computed. The estimated cost forms the basis on which the price negotiations with the customer is made. Other planning activities such as staffing, scheduling, etc., are undertaken based on the estimations made. In subsection 3.3.4, we shall discuss the techniques popularly being used to make these estimations.

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatction and adversely affect team morale. It can even cause project failure. For this reason, ject planning is considered to be a very important activity. However, for effective project nning, in addition to the knowledge of the various estimation techniques, past experience rucial.

Especially for large projects, it becomes very difficult to make accurate plans. A part of s difficulty is due to the fact that the project parameters, scope of the project, project staff, may change during the span of the project. In order to overcome this problem, sometimes ject managers undertake project planning in stages. Planning a project over a number of ges protects managers from making big commitments too early. This technique of staggered nning is known as *Sliding Window Planning*. In the sliding window technique, starting with initial plan, the project is planned more accurately in successive development stages. At start of a project, project managers have incomplete knowledge about the details of the ~~ject. Their information base gradually~~ improves as the project progresses through different ses. After the completion of every phase, the project managers can plan each subsequent se more accurately and with increasing levels of confidence.

.1 The SPMP Document

e project planning is complete, project managers document their plans in a Software ject Management Plan (SPMP) document. Listed below are the different items that the IP document should discuss. This list can be used as a possible organization of the SPMP ment.

ganization of the Software Project Management Plan (SPMP) cument:

. **Introduction**
   - (a) Objectives
   - (b) Major Functions
   - (c) Performance Issues
   - (d) Management and Technical Constraints

Project estimates
   - (a) Historical Data Used
   - (b) Estimation Techniques Used.
   - (c) Effort, Resource, Cost, and Project Duration Estimates

Schedule
   - (a) Work Breakdown Structure
   - (b) Task Network Representation
   - (c) Gantt Chart Representation
   - (d) PERT Chart Representation

Project resources
   - (a) People
   - (b) Hardware and Software
   - (c) Special Resources

5. **Staff organization**
   - (a) Team Structure
   - (b) Management Reporting
6. **Risk management plan**
   - (a) Risk Analysis
   - (b) Risk Identification
   - (c) Risk Estimation
   - (d) Risk Abatement Procedures
7. **Project tracking and control plan**
8. **Miscellaneous plans**
   - (a) Process Tailoring
   - (b) Quality Assurance Plan
   - (c) Configuration Management Plan
   - (d) Validation and Verification
   - (e) System Testing Plan
   - (f) Delivery, Installation, and Maintenance Plan

3.3 METRICS FOR PROJECT SIZE ESTIMATION

As already mentioned, accurate estimation of the problem size is fundamental to satisfactory estimation of other project parameters such as effort, time duration for completing the project and the total cost for developing the software. Before discussing appropriate metrics to estimate the size of a project, let us examine what the term problem size means in the context of software projects. The size of a project is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code.

> The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently, two metrics are popularly being used to estimate size: *lines of code* (LOC) and *function point* (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages which are discussed in the following.

3.3.1 Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines are ignored.

Determining the LOC count at the end of a project is very simple. 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, one would have to make a systematic guess.

Project managers usually divide the problem into modules, and each module into submodules, and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to predict the LOC count for the various leaf-level modules sufficiently accurately, past experience in developing similar products is very helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation. However, LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style — different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted in stead of lines of code.
- LOC is a measure of the coding activity alone. On the other hand, a good problem size measure should consider the total effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.
- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in by different developers (that is, productivity), they would be discouraging code reuse by developers!
- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can only be accurately computed only after the code has been fully developed. Therefore, the LOC metric is of little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

3.3.2 Function Point Metric

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.

The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the query-book feature (see Figure 3.2) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data. Thus, a computation of the number of input and output data values to a system gives some indication of the number of functions supported by the system.

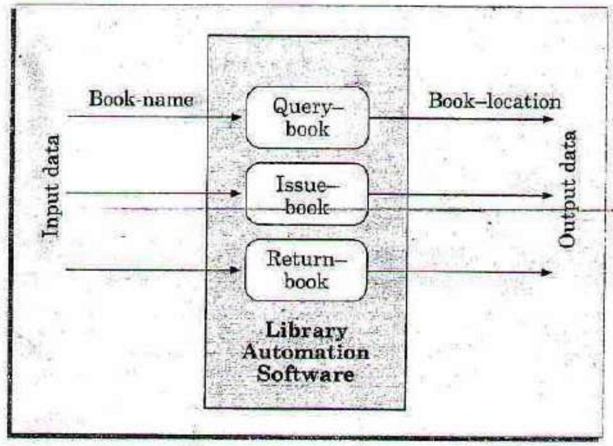

Figure 3.2: System function as a map of input data to output data.

Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces. Interfaces refer to the different mechanisms that need to be supported for data transfer with other external systems. Besides using the number of input and output data values, function point metric computes the size of a software product (in units of function points or FPs) using three other characteristics of the product discussed above and shown in the following expression.

Function point is computed in three steps. The first step is to compute the unadjusted function point (UFP). In the next step, the UFP is refined to reflect the differences in the complexities of the different parameters of the expression for UFP computation (shown below). In the third and the final step, FP is computed by further refining UFP to account for the specific characteristics of the project that can influence the development effort.

```
UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 +
(Number of files)*10 + (Number of interfaces)*10
```

The expression shows the computation of the unadjusted function points (UFP) as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed by Albrecht empirically and was validated through data gathered from many projects.

The meaning of the different parameters of this expression is as follows:

1. Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquires. Inquiries are user commands such as `print-account-balance`. Inquiries are counted separately. It must be noted that individual data items input by the user are not simply added up to compute the number of inputs, but a group of related inputs are considered as a single input. For example, while entering the data concerning an employee to an employee payroll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

2. Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While computing the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one output.

3. Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

4. Number of files: Each logical file is counted. A logical file implies a group of logically related data. Thus, logical files include data structures and physical files.

5. Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.

The computed UFP is refined in the next step. The complexity level of each of the parameters are graded as simple, average, or complex. The weights for the different parameters can then be computed based on Table 3.1. Thus, rather than each input being computed as four function points, very simple inputs can be computed as three function points and very complex inputs as six function points.

Table 3.1: Refinement of function point entities

Type	*Simple*	*Average*	*Complex*
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of various project parameters that can influence the development effort such as high transaction rates, response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort. Each of these 14 factors is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.35. Finally, FP is given as the product of UFP and TCF. Thar is, FP=UFP*TCF.

Feature point metric. A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true. The effort required to develop any two functionalities may vary widely. For example, in a library automation software, the `create-member` feature would be much simpler compared to the `loan-from-remote-library` feature. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of developing the various functionalities. To overcome this problem, an extension of the function point metric called *feature point metric* has been proposed.

Feature point metric incorporates *algorithm* complexity as an extra parameter. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

Proponents of function point and feature point metrics claim that these metrics are language-independent and can be easily computed from the SRS document during project planning, whereas opponents claim that these metrics are subjective and require a sleight of hand. An example of the subjective nature of the function point metric can be that the way one would group logically related data items can be very subjective. For example, consider that certain data `employee-details` consists of the employee name and employee address. Then, it is possible that one can consider it as a single unit of data. Also, someone else can consider the employees address as one unit and name as another. Therefore, there is sufficient scope for different project managers to arrive at different function point measures for the same problem.

3.4 PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration and cost. These estimates not only help in quoting an appropriate project cost to the customer but also form the basis for resource planning and scheduling. There are three broad categories of estimation techniques:

1. Empirical estimation techniques
2. Heuristic techniques
3. Analytical estimation techniques

In the following, we provide an overview of the different categories of estimation techniques.

3.4.1 Empirical Estimation Techniques

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, over the years, different activities involved in estimation have been formalized to certain extent. We shall discuss two such formalizations of the basic empirical estimation techniques in sections 3.5.1 and 3.5.2.

3.4.2 Heuristic Techniques

Heuristic techniques assume that the relationships among the different project parameters can be modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and multivariable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

$$\text{Estimated Parameter} = c_1 * e^{d_1}$$

In the above expression, e is a characteristic of the software which has already been estimated (independent variable). *Estimated parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc. c_1 and d_1 are constants. The values of the constants c_1 and d_1 are usually determined using data collected from past projects (historical data). The COCOMO model (discussed in section 3.6.1), is an example of a single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\text{Estimated resource} = c_1 * ep_1^{d_1} + c_2 * ep_2^{d_2} + ...$$

where ep_1, ep_2, ... are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, ... are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the constants c_1, c_2, d_1, d_2, ... Values of these constants are usually determined from historical data. The intermediate COCOMO model discussed in section 3.6.2 can be considered to be an example of a multivariable estimation model.

3.4.3 Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis. We shall discuss the Halstead's software science in Section 3.7 as an example of an analytical technique. As we will see, Halstead's software science can be used to derive some interesting results, starting with a few simple assumptions. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

3.5 EMPIRICAL ESTIMATION TECHNIQUES

We have already pointed out that empirical estimation techniques have over the years been formalized to certain extent, yet these are still essentially euphemisms for pure guess work. Two popular empirical estimation techniques are: expert judgement and Delphi estimation techniques.

3.5.1 Expert Judgement Technique

Expert judgement is one of the most widely used estimation techniques. In this technique, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgement is the estimation made by a group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different developers. The work breakdown structure formalism discussed in section 3.9.1 helps the manager to breakdown the tasks systematically.

After the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. In general, the task dependencies define a partial ordering among tasks, i.e. each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). Let us discuss the dependency among the activities are represented in the form of an activity network.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT (Project Evaluation and Review Technique) chart representation is developed. The PERT chart representation is suitable for project monitoring and control. The work breakdown structure, activity network, Gantt and PERT charts are discussed further. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called a **milestone.** The project manager tracks the progress of a project by monitoring the the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

3.9.1 Work Breakdown Structure

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. The root of the tree is labelled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. Figure 3.7 represents the WBS of an MIS (Management Information System) software.

While breaking down a task into smaller tasks, 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 developers. If the activity ordering permits that solutions to these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower of course!). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

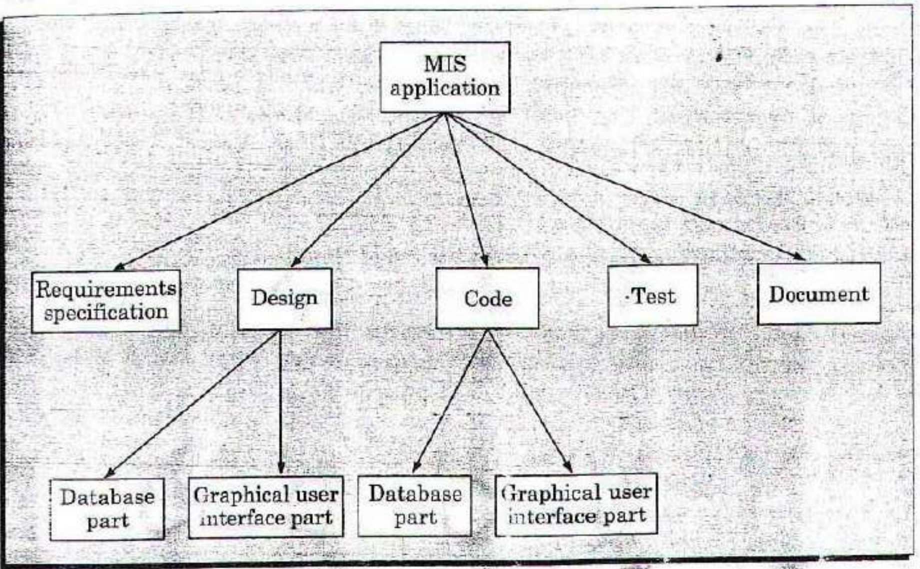
MIS application
Requirements specification
Design
Code
Test
Document
Database part
Graphical user interface part
Database part
Graphical user interface part

Figure 3.7: Work breakdown structure of an MIS problem.

3.9.2 Activity Networks and Critical Path Method

WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies. Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however is not a good idea, because software developers often resent such unilateral decisions. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the developers to do a better and faster job. On the other hand, careful experiments have shown that unrealistically aggressive schedules not only cause developers to compromise on intangible quality aspects, but also are a cause for schedule delays. A possible alternative is to let each engineer himself estimate the time for an activity he can be assigned to. This approach can help to accurately estimate the task durations without creating undue schedule pressures.

Critical Path Method (CPM)

From the activity network representation, 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 time is the difference between MT and the maximum of all paths from this 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 that a task may be delayed before it will affect the end 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. Thus, any path whose duration equals MT is a critical path. As a result, there can be more than one critical path for a project.

The project parameters for different tasks for the MIS problem is shown in Table 3.6.

Table 3.6: Project parameters computed from activity network

Task	*ES*	*EF*	*LS*	*LF*	*ST*
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code data base	60	165	60	165	0
Code GUI part	45	90	120	165	75
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT. The critical path in Figure 3.8 is shown with a thick arrow.

3.9.3 Gantt Charts

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Gantt charts used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software 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. A Gantt chart representation for the MIS problem of Figure 3.8 is shown in Figure 3.9.

3.9.4 PERT Charts

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually annotated

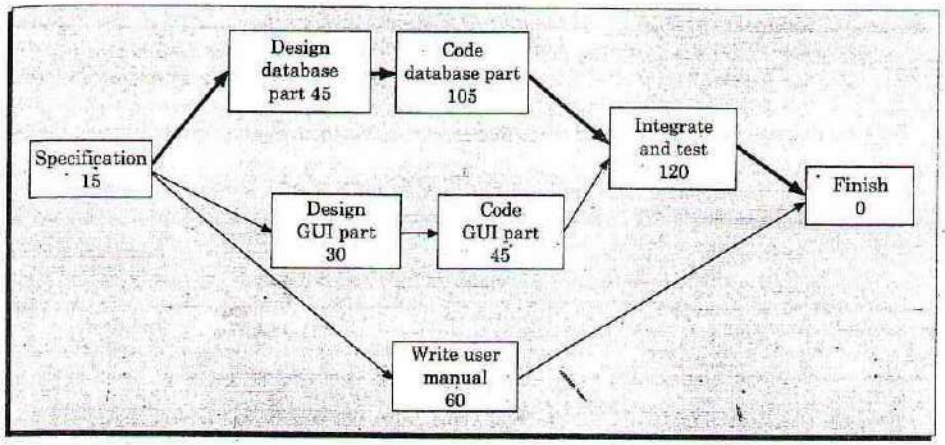

Figure 3.8: Activity network representation of the MIS problem.

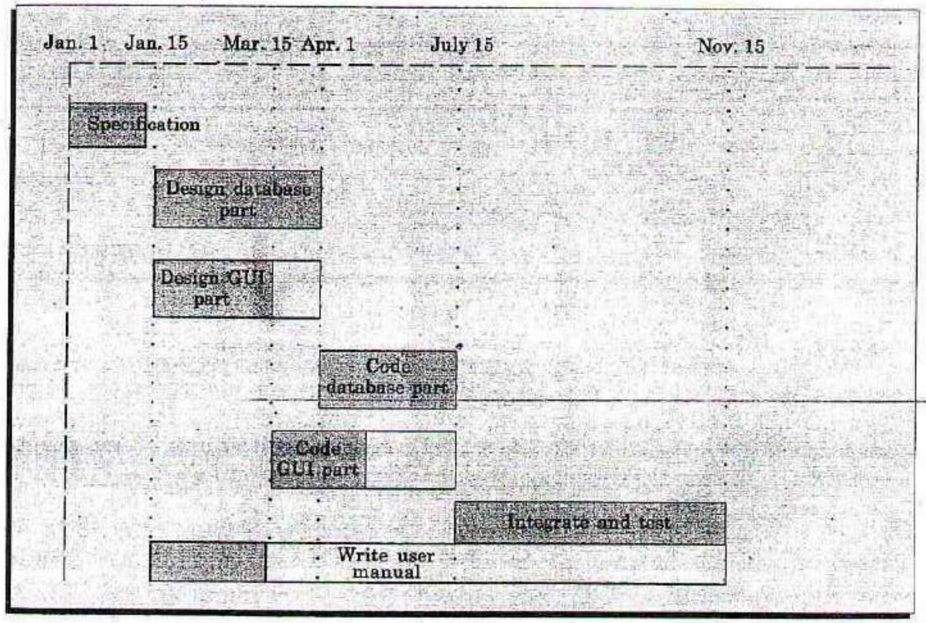

Figure 3.9: Gantt chart representation of the MIS problem.

with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there is not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of Figure 3.8 is shown in Figure 3.10. PERT charts are a 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 durations, the utility of the activity diagrams are limited.

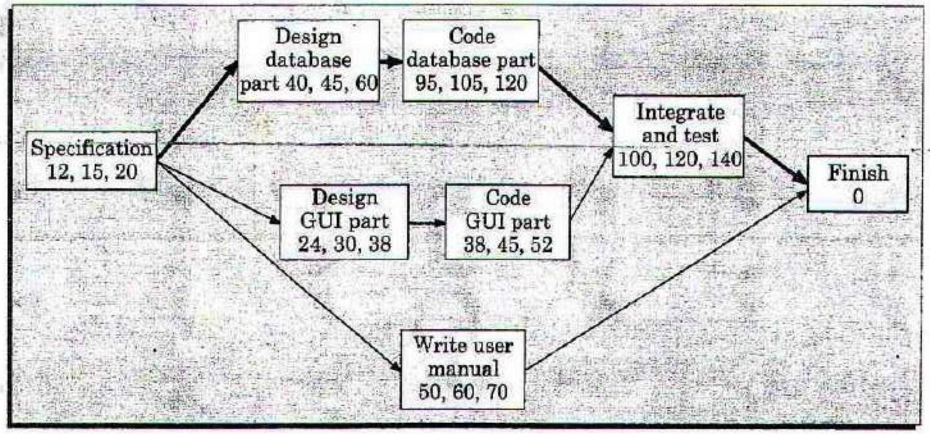
Specification 12, 15, 20
Design database part 40, 45, 60
Code database part 95, 105, 120
Integrate and test 100, 120, 140
Finish 0
Design GUI part 24, 30, 38
Code GUI part 38, 45, 52
Write user manual 50, 60, 70

Figure 3.10: PERT chart representation of the MIS problem.

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different developers.

3.9.5 Project Monitoring and Control

Once the project gets underway, the project manager has to monitor the project continuously to ensure that it is progressing as per the plan. The project manager designates certain key events such as completion of some important activity as **milestones**. For example, a milestone can be the completion and review of the SRS document, completion of the coding and unit testing, etc. Once a milestone is reached, the project manager can assume that some measurable progress has been made. If any delay in reaching a milestone is predicted, then corrective actions might have to be taken. This may entail reworking all the schedules and producing a fresh schedule.

As already mentioned, the PERT charts are especially useful in project monitoring and control. A path in this graph is any set of consecutive nodes and edges in this graph from the starting node to the last node. A **critical path** in this graph is a path along which every milestone is critical to meeting the project deadline. In other words, if any delay occurs along a critical path, the entire project would get delayed. It is, therefore, necessary to identify all the critical paths in a schedule—adhering to the schedules of the tasks appearing on the critical paths is of prime importance to meet the delivery date. Please note that there may be

more than one critical path in a schedule. The tasks along a critical path are called *critical tasks*. If necessary, a manager may switch resources from a noncritical task to a critical task so that all milestones along the critical path are met.

Several tools are available which can help you to figure out the critical paths in an unrestricted schedule, but figuring out an optimal schedule with resource limitations and with a large number of parallel tasks is a very hard problem. There are several commercial products for automating the scheduling techniques are available. Popular tools to help draw the schedule-related graphs include the MS-Project software available on personal computers.

3.10 ORGANIZATION AND TEAM STRUCTURES

Usually, every software development organization handles several projects at any time. Software organizations assign different teams of developers to handle different software projects. Thus, there are two important issues: How is the organization as a whole structured? And, how are the individual project teams structured? There are a few standard ways in which software organizations and teams can be structured.

3.10.1 Organization Structure

There are essentially two broad ways in which a software 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 for (see Figure 3.11). In the functional format, the development staff are divided based on the functional group to which they belong to. The different projects borrow developers from functional groups for specific phases of the project and return them to the functional group upon the completion of the phase.

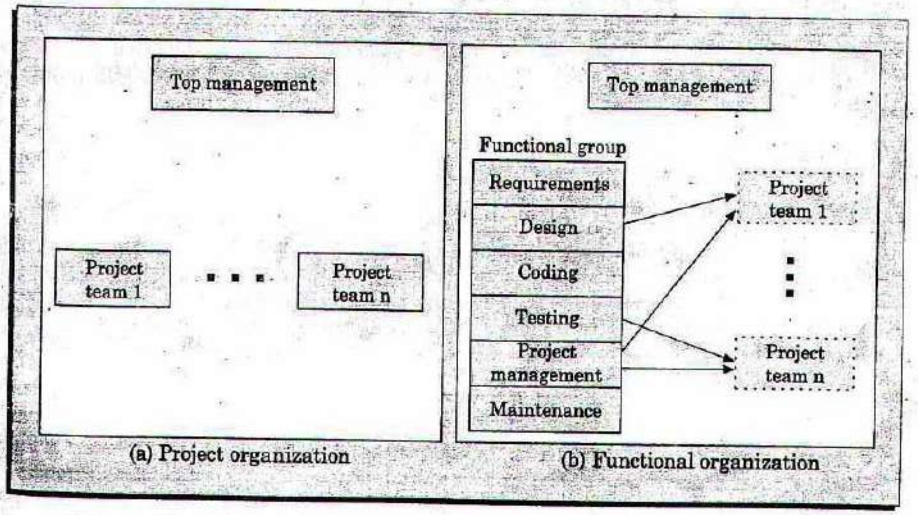

Figure 3.11: Schematic representation of the functional and project organization.

conception held by managers as evidenced in their staffing, planning and scheduling practices, is the assumption that one software engineer is as productive as another. However, experiments have revealed that there exists a large variability of productivity between the worst and the best software developers in a scale of 1 to 30. In fact, the worst developers may sometimes even reduce the overall productivity of the team, and thus in effect exhibit negative productivity. Therefore, choosing good software developers is crucial to the success of a project.

3.11.1 Who is a Good Software Engineer?

In the past, several studies concerning the traits of a good software engineer have been carried out. All these studies roughly agree on the following attributes that good software developers should possess:

1. Exposure to systematic techniques, i.e. familiarity with software engineering principles
2. Good technical knowledge of the project areas (*domain knowledge*)
3. Good programming abilities
4. Good communication skills like oral, written and interpersonal skills
5. High motivation
6. Sound knowledge of fundamentals of computer science
7. Intelligence
8. Ability to work in a team
9. Discipline

Studies show that these attributes vary as much as 1:30 for poor and bright candidates. An experiment conducted by Sackman [1968] shows that the ratio of coding hour for the worst to the best programmers is 25:1, and the ratio of debugging hours is 28:1. Also, the ability of a software engineer to arrive at the design of the software from a problem description varies greatly with respect to the parameters of quality and time.

Technical knowledge in the area of the project (*domain knowledge*) is an important factor determining the productivity of an individual for a particular project, and the quality of the product that he develops. A programmer having a thorough knowledge of database applications (e.g. MIS) may turn out to be a poor data communication developer. Lack of familiarity with the application areas can result in low productivity and poor quality of the product.

Since software development is a group activity, it is vital for a software developer to possess three main kinds of communication skills: oral, written and interpersonal. A software developer not only needs to effectively communicate with his teammates (e.g. reviews, walk throughs, and other team communications) but may also have to communicate with the customer to gather product requirements. Poor interpersonal skills hamper these vital activities and often show up as poor quality of the product and low productivity. Software developers are also required at times to make presentations to the managers and to the customers. This requires a different kind of communication skill (oral communication skill). A software developer is also expected to document his work (design, code, test, etc.) as well as write the users' manual, training manual, installation manual, maintenance manual, etc. This requires good *written communication* skill.

Motivation level of software developers is another crucial factor contributing to his work quality and productivity. Even though no systematic studies have been reported in this regard,

it is generally agreed that even bright developers may turn out to be poor performers when they lack motivation. An average developer who can work with a single mind track can outperform other developers. But motivation is a complex phenomenon requiring careful control. For majority of software developers, higher incentives and better working conditions have only limited affect on their motivation levels. Motivation is to a great extent determined by personal traits, family and social backgrounds, etc.

3.12 RISK MANAGEMENT

A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway. We should distinguish between a risk which is a problem that might occur from the problems currently being faced by a company. If a risk becomes true, the anticipated problem becomes a reality and is no more a risk. If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project. Therefore, it is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that contingency plans can be prepared to contain each risk. In this context, risk management aims at reducing the impact of all kinds of risks that might affect a project. Risk management consists of three essential activities: risk identification, risk assessment, and risk containment. We discuss these three activities in the following subsections.

3.12.1 Risk Identification

The project manager needs to anticipate the risks in the project as early as possible so that the impact of the risks can be minimized by making effective risk management plans. So, early risk identification is important. Risk identification is somewhat similar to listing down your nightmares. For example, you might be worried whether the vendors whom you have asked to develop certain modules might not complete their work in time, whether they would turn in poor quality work, whether some of your key personnel might leave the organization, etc. All such risks that are likely to affect a project must be identified and listed.

A project can be affected by a large variety of risks. In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorize risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project as follows:

Project risks

Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. For any manufacturing project such as manufacturing of cars the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus, he can easily assess the progress of the work and control it. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

Technical risks

Technical risks concern potential design, implementation, interfacing, testing and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due the development team's insufficient knowledge about the product.

Business risks

This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments.

Example classification of risks in a project

Let us consider the satellite based mobile communication product which we considered in Section 2.6 of Chapter 2. The project manager can identify several risks in this project. We can classify them appropriately as:

1. What if the project cost escalates to a large extent than what was estimated?—Project risk.
2. What if the mobile phones becomes too large for people to conveniently carry?—Business risk
3. What if it is later found out that the level of radiation is harmful to human being?—Business risk
4. What if hand off between satellites becomes too difficult to implement?—Technical risk

In order to be able to successfully foresee and identify different risks that might affect a software project, it is a good idea to have a **company disaster list**. This list would contain all the bad events that have happened to software projects of the company over the years including events that can be laid at the customer's doors. This list can be read by the project mangers in order to be aware of some of the risks that a project might be susceptible to. Such a disaster list has been found to help in performing better risk analysis.

3.12.2 Risk Assessment

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

1. The likelihood of a risk coming true (r).
2. The consequence of the problems associated with that risk (s).

Based on these two factors, the priority of each risk can be computed:

$$p = r * s$$

where, p is the priority with which the risk must be handled, r is the probability of the risk becoming true, and s is the severity of damage caused due to the risk becoming true. If all identified risks are prioritized, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for these risks.

3.12.3 Risk Containment

After all the identified risks of a project are assessed, plans must be made to first 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 the risks. There are three main strategies to plan for risk containment:

Avoid the risk

Risks can be avoided in several ways, such as discussing with the customer to change the requirements to to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover.

Transfer the risk

This strategy involves getting the risky component developed by a third party, buying insurance cover, and so on.

Risk reduction

This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned. The most important thing to do in addressing technical risks is to build a prototype that tries out pieces of the technology that you are trying to use. For example, if you are using a compiler for identifying commands in the user interface, you would have to construct a compiler for a small language first.

To choose between the different strategies of handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this, we may compute the **risk leverage** of the different risks. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

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

Even though we identified three broad ways to handle any risk, but still risk handling requires a lot of ingenuity on the part of the project manager. As an example, let us consider the options available to contain an important type of risk that occurs in many software projects — that of schedule slippage. Risks relating to schedule slippage arise primarily due to the intangible nature of software. For a project such as building a house, the progress can easily be seen and assessed by the project manager. If he finds that the project is lagging behind then corrective actions can be initiated. Considering that software development per se is invisible, the first step in managing the risks of schedule slippage, is to increase the visibility of the software product. Visibility of a software product can be increased by producing relevant documents during the development process wherever meaningful, and getting these documents reviewed by an appropriate team. Milestones should be placed at regular intervals through a software engineering process to provide a manager with regular indication of progress. Completion of a phase of the development process being followed need not be the only milestones. Every phase can be broken down to reasonable-sized tasks and milestones can be scheduled for these tasks too. A milestone is reached, once documentation produced as part of a software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to set a milestone every 10 to 15 days.

3.13 SOFTWARE CONFIGURATION MANAGEMENT

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g. source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software developers throughout the life cycle of the software. The state of all these objects at any point of time is called the *configuration* of the software product. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

Software configuration management deals with effectively tracking and controlling the configuration of a software product during its life cycle.

Before we discuss configuration management, we must be clear about the distinction between a version and a revision of a software product. A new version of a software is created when there is significant change in functionality, technology, or the hardware it runs on, etc. On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc. Even the initial delivery might consist of several versions and more versions might be added later on. For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows, and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new **release** of software is an improved system intended to replace an old one. Often systems are described as version m, release n; or simple m.n. Formally, a history relation is version of can be defined between objects. This relation can be split into two subrelations is revision of and is variant of. In the following, we first discuss the necessity of configuration management and subsequently we discuss the configuration management activities and tools.

3.13.1 Necessity of Software Configuration Management

There are several reasons for putting an object under configuration management. But, possibly the most important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems can appear. The following are some of the important problems that can appear if configuration management is not used.

Inconsistency problem when the objects are replicated

Consider a scenario where every software developer has a personal copy of an object (e.g. source code). As each developer makes changes to his local copy, he is expected to intimate the changes that he has made to other developer, so that the changes in interfaces are uniformly changed across all modules. However, many times a developer makes changes to the interfaces in his local copies and forgets to intimate other teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the product is in-

tegrated, it does not work. Therefore, when several team members work on developing an object, it is necessary for them to work on a single copy of the object, otherwise inconsistency may arise.

Problems associated with concurrent access

Assume that only a single copy of a program module is maintained, and several developers are working on it. Two developers may simultaneously carry out changes to the different portions of the same module, and while saving overwrite each other. Though we explained the problem associated with concurrent access to program code, similar problems can occur for any other deliverable object.

Providing a stable development environment

When a project work is underway, the team members need a stable environment to make progress. Suppose one is trying to integrate module A, with the modules B and C. He cannot make progress if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces recompilation of module A. When an effective configuration management is in place, the manager freezes the objects to form a *base line*. When any one needs to change any of the objects under configuration control, he is provided with a copy of the base line item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new base line gets formed instantly. This establishes a baseline for others to use and depend on. Also, baselines may be archived periodically (Archiving means copying to a safe place such as a magnetic tape).

System accounting and maintaining status information

System accounting keeps track of who made a particular change to an object and when the change was made.

Handling variants

Existence of variants of a software product causes some peculiar problems. Suppose you have several variants of the same module, and find a bug in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, you should not have to fix it in each and every version and revision of the software separately.

3.13.2 Configuration Management Activities

Normally, a project manager performs the configuration management activity by using an automated configuration management tool. A configuration management tool provides automated support for overcoming all the problems mentioned above. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the developer to change 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.

CHAPTER 11

SOFTWARE RELIABILITY AND QUALITY MANAGEMENT

Reliability of a software product is an important concern for most users. Users not only want the products they purchase to be highly reliable, but for certain categories of products they may even require a quantitative guarantee on the reliability of the product before making their buying decision. This may especially be true for safety-critical and embedded software products. However, as we discuss in this chapter, it is very difficult to accurately measure the reliability of any software product. One of the main problems encountered while quantitatively measuring the reliability of a software product is the fact that reliability is observer-dependent. That is, different groups of users may arrive at different reliability estimates for the same product. Besides this, several other problems (such as frequently changing reliability values due to bug corrections) make accurate measurement of the reliability of a software product difficult. We investigate these issues in this chapter. Even though no metric to accurately measure the reliability of a software product exists, we shall discuss some metrics that are being used at present to quantify the reliability of a software product. We shall also address the problem of reliability growth modelling and examine how to predict when (and if at all) a given level of reliability will be achieved. We shall also examine the statistical testing approach to reliability estimation. In this chapter, we shall also discuss various issues associated with Software Quality Assurance (SQA).

Software Quality Assurance (SQA) has emerged as one of the most talked about topics in recent years in software industry circles. The major aim of SQA is to help an organization develop high quality software products in a repeatable manner. A software development organization can be called **repeatable** when its software development process is person-independent. That is, the success of a project does not depend on who exactly are the team members of the project. Besides, the quality of the developed software and the cost of development are important issues addressed by SQA. In this chapter, we first discuss a few important issues concerning software reliability measurement and prediction before starting our discussion on software quality assurance.

11.1 SOFTWARE RELIABILITY

The reliability of a software product essentially denotes its **trustworthiness or dependability**. Alternatively, the reliability of a software product can also be defined as the probability of the product working correctly over a given period of time.

Intuitively, it is obvious that a software product having a large number of defects is unreliable. It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced. It would have been very nice if we could mathematically characterize this relationship between reliability and the number of bugs present in the system using a simple closed form expression. Unfortunately, it is very difficult to characterize the observed reliability of a system in terms the number of latent defects in the system using a simple mathematical expression. To get an insight into this issue, consider the following. Removing errors from those parts of a software product that are very infrequently executed, makes little difference to the perceived reliability of the product. 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. The most used 10% instructions are often called the *core*[1] of a program. The rest 90% of the program statements are called **non-core** and are on the average 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 result in 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 instruction having the error is executed. If an error is removed from an instruction that is frequently executed (i.e. belonging to the core of the program), then this would show up as a large improvement to the reliability figure. On the other hand, removing errors from parts of the program that are rarely used, may not cause any appreciable change to the reliability of the product.

Based on the above discussion, we can say that 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 on how the product is used, or on its execution profile. If the users execute only those features of a program that are correctly implemented, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low. Different categories of users of a software product typically execute different functions of a software product. For example, for a library automation software the library members would use functionalities such as issue book, search book, etc. on the other hand, the librarian would normally execute features such as create member, create book record, delete member record, etc. So defects which show up for the librarian, may not show up for the members. Suppose the functions of a Library Automation Software which the library members use are error-free; and functions used by the librarian have many bugs. Then, these two categories of users would have very different opinions about the reliability of the software. Therefore, the reliability figure of a software product is observer-dependent, and it is very difficult to absolutely quantify the reliability of the product.

[1]To determine the core and non-core parts of a program, you can use a commonly available tool called a profiler. On Unix platforms, a tool called **prof** is normally available for this purpose.

Based on the above discussions, we can summarize the main reasons that make software reliability more difficult to measure than hardware reliability:

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

In the following subsection, we shall discuss why software reliability measurement is a harder problem than hardware reliability measurement.

11.1.1 Hardware versus Software Reliability

An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.

> Hardware components fail due to very different reasons as compared to software components. Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.

A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part. In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug. For this reason, when a hardware part is repaired, its reliability would be maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug fix introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, the inter-failure times remain constant). On the other hand, the aim of software reliability study would be reliability growth (that is, increase in inter-failure times).

A comparison of the changes in failure rate over the product lifetime for a typical hardware product as well as a software product are sketched in Figure 11.1. Observe that the plot of change of reliability with time for a hardware component [Figure 11.1(a)] appears like a bath tub. For a software component the failure rate is initially high, but decreases as the faulty components are identified are either repaired or replaced. The system then enters its useful life, where the rate of failure is almost constant. After some time (called product lifetime) the major components wear out, and the failure rate increases. The initial failures are usually covered through manufacturer's warranty. A corollary of this observation (though a digression from our topic of discussion) is that it may be unwise to buy a product (even at a good discount to its face value) towards the end of its lifetime. That is, one need not feel happy to buy a ten year old car at one tenth of the price of a new car, since it would be near the rising edge of the bath tub curve, and one would have to spend unduly large time, effort, and money on repairing and end up as the loser. In contrast to the hardware products, the software product show the highest failure rate just after purchase and installation [see the initial portion of the plot in Figure 11.1 (b)]. As the system is used, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the

useful life of the product. As the software becomes obsolete no more error correction occurs and the failure rate remains unchanged.

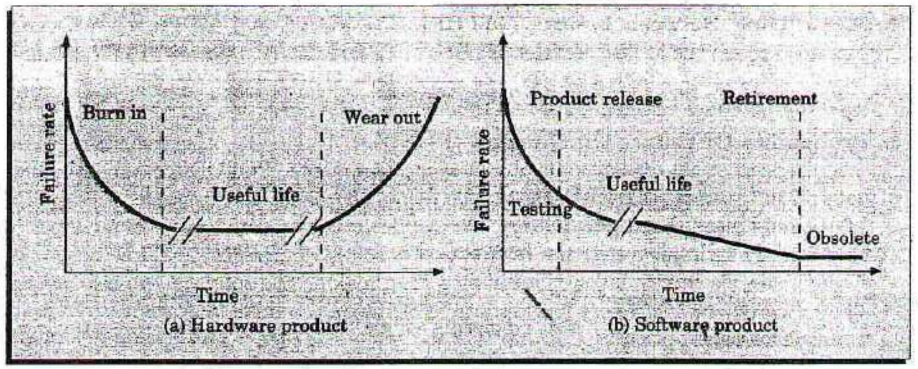

Figure 11.1: Change in failure rate of a product.

11.1.2 Reliability Metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. However, in practice, it is very difficult to formulate a metric using which precise reliability measurement would be possible. In the absence of such measures, we discuss six metrics that correlate with reliability as follows:

1. **Rate of OCcurrence Of Failure (ROCOF).** ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation. However, many software products do not run continuously (unlike a car or a mixer), but deliver certain service when a demand is placed on them. For example, a library software is idle until a book issue request is made. Therefore, for a typical software product such as a payroll software, applicability of ROCOF is very limited.
2. **Mean Time To Failure (MTTF).** MTTF is the time between two successive failures, averaged 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 $t_1, t_2, ..., t_n$. Then, MTTF can be calculated as $\sum_{i=1}^{n} \frac{t_{i+1}-t_i}{(n-1)}$. It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc. are not taken into account in the time measurements and the clock is stopped at these times.

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 to fix them.

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

5. **Probability Of Failure On Demand (POFOD).** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. 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. We have already mentioned that the reliability of a software product should be determined through specific service invocations, rather than making the software run continuously. Thus, POFOD metric is very appropriate for software products that are not required to run continuously.

6. **Availability.** Availability of a system is a measure of how likely would 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 (down time) of a system when a failure occurs. Th[illegible] metric is important for systems such as telecommunication systems, and operating systems, and embedded controllers, etc. which are supposed to be never down and where repair and restart time are significant and loss of service during that time cannot be overlooked.

All the above reliability metrics suffer from several shortcomings as far as their use in software reliability measurement is concerned. One of the reasons is that these metrics are centred around the probability of occurrence of system failures, but take no account of the consequences of failures. That is, these reliability models do not distinguish the relative severity of different failures. Failures which are transient and whose consequences are not serious are in practice of little concern in the operational use of a software product. These types of failures can at best be minor irritants. On the other hand, more severe types of failures may render the system totally unusable. In order to estimate the reliability of a software product more accurately, it is necessary to classify various types of failures. Please note that the different classes of failures may not be mutually exclusive. The classification is based on widely different set of criteria. As a result, a failure type can at the same time belong to more than one class. A scheme of classification of failures is as follows:

1. **Transient:** Transient failures occur only for certain input values while invoking a function of the system.

2. **Permanent:** Permanent failures occur for all input values while invoking a function of the system.

3. **Recoverable:** When a recoverable failure occurs, the system can recover without having to shutdown and restart the system (with or without operator intervention).

4. **Unrecoverable.** In unrecoverable failures, the system may need to be restarted.

5. **Cosmetic:** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the situation where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

11.1.3 Reliability Growth Modelling

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired.

A reliability growth model can be used to predict when (or if at all) 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, in this text we will discuss only two very 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. Such a model is shown in Figure 11.2. 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 correction of different errors contribute differently to reliability growth.

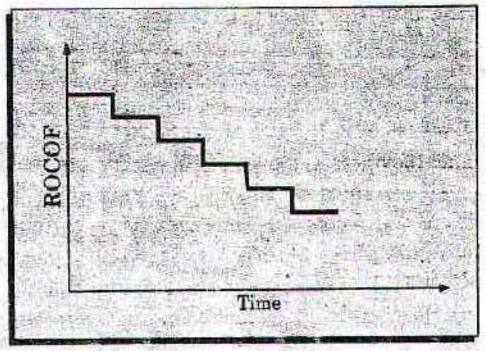

Figure 11.2: Step function model of reliability growth.

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 to the product reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

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

11.2 STATISTICAL TESTING

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

Operation profile

Different categories of users may use a software product for very different purposes. For example, a librarian might use the library automation software to create member records, delete member records, add books to the library, etc; whereas a library member might use to software to query about the availability of a book, and to issue and return books. Formally, we can define the operation profile of a software as the probability of a user selecting the different functionalities of the software. If we denote the set of various functionalities offered by the software by $\{f_i\}$, the operational profile would associate with each function $\{f_i\}$ with the probability with which an average user would select $\{f_i\}$ as his next function to use. Thus, we can think of the operation profile as assigning a probability value p_i to each functionality f_i of the software.

How to define the operation profile for a product?

We need to divide the input data into a number of input classes. For example, for a graphical editor software, we might divide the input into data associated with the edit, print, and file operations. We then need to assign a probability value to each input class; to signify the probability for an input value from that class to be selected. The operation profile of a software product can be determined by observing and analyzing the usage pattern of the software by a number of users.

Steps in statistical testing

The first step is to determine the operation profile of the software. 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 software 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 to be included. Now let us discuss the implications of these assumptions.

It is straightforward to generate test cases for the common types of inputs, since one can easily write a test case generator program which can automatically generate these test cases. However, it is also required that a statistically significant percentage of the unlikely inputs should also be included in the test suite. Creating these unlikely inputs using a test case generator is very difficult.

Pros and cons of statistical testing

Statistical testing 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 can find to be more reliable (than actually it is!). Also, the reliability estimation arrived by using statistical testing is more accurate compared to those of other methods discussed. However, it is not easy to perform the statistical testing satisfactorily due to the following two reasons. 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.

11.3 SOFTWARE QUALITY

Traditionally, the quality of a product is defined in terms of its fitness of purpose. That is, a good quality product does exactly what the users want it to do, since for almost every product, fitness of purpose is interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although fitness of purpose is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc.—fitness of purpose is not a wholly satisfactory definition of quality for software products. To give an example of why this is so, consider a software product that is functionally correct. That is, it correctly performs all the functions that have been specified in its SRS document. Even though it may be functionally correct, we cannot consider it to be a quality product, if it has an almost unusable user interface. Another example is that of a product which does everything that the users wanted, but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as fitness of purpose for software products is not wholly satisfactory.

The modern view of a quality associates with a software product several quality factors such as the following:

1. Portability: A software product is said to be portable, if it can be easily made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.

2. Usability: A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product.

3. Reusability: A software product has good reusability, if different modules of the product can easily be reused to develop new products.

4. Correctness: A software product is *correct*, if different requirements as specified in the SRS document have been correctly implemented.

5. Maintainability: A software product is *maintainable*, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

11.4 SOFTWARE QUALITY MANAGEMENT SYSTEM

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. In the following, we briefly discuss some of the important issues associated with a quality system.

- A quality system is the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.
- The quality system activities encompass the following:
  - Auditing of 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.

11.4.1 Evolution of Quality Systems

Quality systems have rapidly evolved over the last six decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. For example, a company manufacturing nuts and bolts would inspect its finished goods and would reject those nuts and bolts that are outside certain specified tolerance range. Since that time, quality systems of organizations have undergone four stages of evolution as shown in Figure 11.3. The initial product inspection method gave way to quality control (QC) principles.

Quality Control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

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 the Quality Assurance (QA) Principles.

The basic premise of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality.

The modern quality assurance paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must continuously be improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes 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, whereas our focus in this text is reengineering of the software development process. From the above discussion, we can say that over the last six decades or so, the quality paradigm has shifted from product assurance to process assurance (see Figure 11.3).

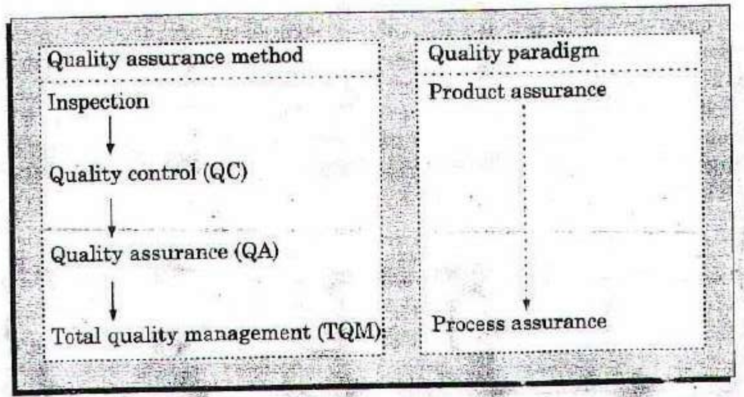

Figure 11.3: Evolution of quality system and corresponding shift in the quality paradigm.

11.4.2 Product Metrics versus Process Metrics

All modern quality systems lay emphasis on collection of certain product and process metrics during product development. Let us first understand the basic differences between product and process metrics.

Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.

Examples of product metrics are LOC and function point to measure size, PM (person-month) to measure the effort required to develop it, months to measure the time required to develop the product, time complexity of the algorithms, etc. Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, number of latent defects per line of code in the developed product.

11.5 ISO 9000

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

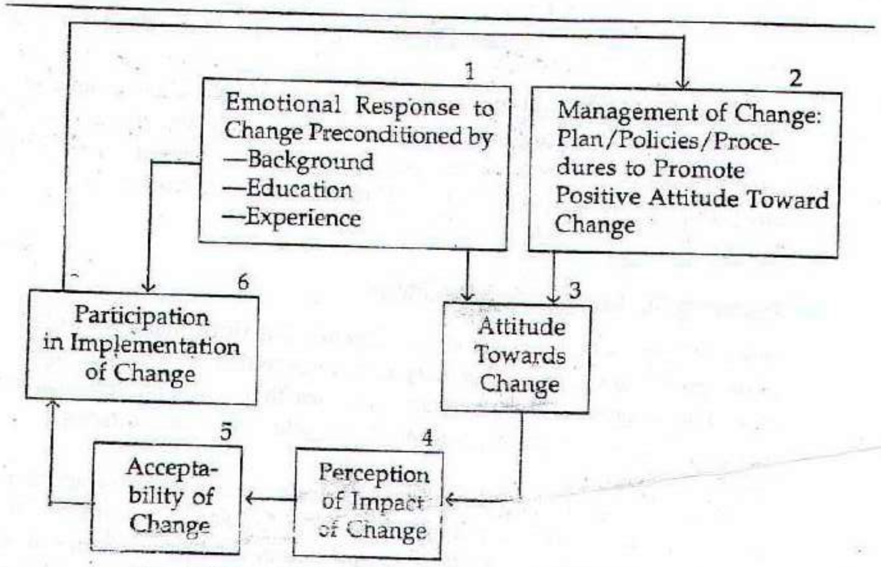
1
Emotional Response to Change Preconditioned by
—Background
—Education
—Experience
2
Management of Change: Plan/Policies/Procedures to Promote Positive Attitude Toward Change
3
Attitude Towards Change
4
Perception of Impact of Change
5
Acceptability of Change
6
Participation in Implementation of Change

Fig. 16.5 Cycle in Attitude Towards Change

Change Strategies

Training and orientation, described earlier in this chapter, are a key to favourable attitudes towards change.

Other ways for management to smooth transition to new information technology are to:

- Assign responsibility for change to upper level managers who possess the organizational power to legitimize change.
- Identify individuals in the organization who must learn new behaviours, skills, or knowledge because of the change and schedule training for them.
- Involve resisters in the development of new systems by giving them an active role in identifying problems and planning solutions.
- Open lines of communication between employees and management. For example, provide forums where employees can voice their concerns about the new technology.
- Avoid secrecy about the new system. Publicize information regarding system changes.
- Pace conversion to allow a readjustment period to the new system.
- Implement new systems in modules.
- Alter job titles to reflect increased responsibility.
- Reward ideas that will improve throughput.
- Document standards so that new procedures are easy to learn and reference.
- Clearly establish in advance the demarcations of authority that will exist following changeover.
- Upgrade the work environment following change, incorporating recommendations of human factors studies.
- Show sympathy and be receptive to complaints following conversion.
- Give job counselling.
- Arrange job transfers.
- Call a hiring freeze until all displaced personnel are reassigned.
- Provide separation pay.
- Organize group therapy.
- Initiate morale-boosting activities, such as a company newsletter and social events.

The key to success of these strategies lies in management's ability to demonstrate support for employees adversely affected by the change. Management should also exhibit patience and understanding when the disruption and dislocation of change produces anxiety and tension even among those not directly involved with the new system. It is largely a skill in handling interpersonal relationships that determines whether a firm can absorb technological advances in the field of computing.

TESTING

Testing is a quality control measure. The information system under development, or parts of the system, are exercised to see if the results satisfy performance specifications and match anticipated results. If not, the reason(s) are traced, modifications are made, and the system is retested—a cycle that is repeated until testers are satisfied with quality.

The underlying premise of testing is that correct results can be predicted and that the validity of a system can be ascertained by comparing results generated during a test against predicted results. Key elements of the system are identified, then duplicated and checked in a production environment that matches the environment in which the system will operate once conversion takes place. Test cases are selected and test data are generated to simulate live, full-sized files and transaction volume. If planning lacks thoroughness and organization so that key elements of the system are not checked at some point, the system may run successfully during the test cycle yet fail when placed in operation. The problem is that a complete definition of test cases is virtually impossible and that exhaustive testing can pass the limits of practicality.

What elements must be checked?

- The hardware on which the new system will operate.
- The operating system that will be used.
- The assembler or compiler that will produce the object code.
- The data to be processed.
- Applications software.
- Data entry methods.
- Operating procedures.
- Output interpretation.

In general, systems are checked for reliability, effectiveness, file integrity,

11.5.1 What is ISO 9000 Certification?

ISO 9000 certification serves as a reference for contract between independent parties. In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether has the vendor has obtained ISO 90000 certification or not. In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system. We have already seen that the quality system of an organization applies to all its activities related to its products or services. The ISO standard addresses both operational aspects (that is, the process) and organizational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 specifies a set of recommendations for repeatable and high quality product development. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003.

The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.

The types of software companies to which the different ISO standards apply are as follows.

1. 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 software development organizations.

2. ISO 9002: This standard applies to those organizations which do not design products but are only involved in production. Examples 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 software development organizations.

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

11.5.2 ISO 9000 for Software 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 the ISO 9000 documents are written using generic terminologies, and it is very difficult to interpret them in the context of software development organizations. An important reason behind such a situation is the fact that software development is in many respects radically different from the development of other types of products. Two major differences between software development and development of other kinds of products are as follows:

- Software is intangible and therefore difficult to control. It means that software would not be visible to the user until the development is complete and the software is up and running. It is difficult to control and manage anything that you cannot see and feel. In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to accurately determine how much work has been completed and to estimate how much more time will it take.

- During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product. As an example, consider a steel making company. The company would consume large amounts of raw material such as iron ore, coal, lime, manganese, etc. Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organizations.

Due to such radical differences between software and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of software industry. Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for software industry. At present, official guidance is inadequate regarding the interpretation of various clauses of ISO 9000 standard in the context of software industry and one has to keep on cross referencing the ISO 9000-3 document.

11.5.3 Why Get ISO 9000 Certification?

There is a mad scramble among software development organizations for obtaining ISO certification due to the benefits it offers. Let us examine some of the benefits that accrue to organizations obtaining ISO certification.

- Confidence of customers in an organization increases when the organization qualifies for ISO 9001 certification. This is especially true in the international market. In fact, many organizations awarding international software development contracts insist that the development organization have ISO 9000 certification. For this reason, it is vital for software organizations involved in software export to obtain ISO 9000 certification.
- ISO 9000 requires a well-documented software production process to be in place. A well-documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient and cost-effective.
- ISO 9000 certification points out the weak points of an organizations and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and TQM.

11.5.4 How to Get ISO 9000 Certification?

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

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

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

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

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

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

6. Continued surveillance: The registrar continues monitoring the organization periodically.

ISO mandates that a certified organization can use the certificate for corporate advertisements but cannot use the certificate for advertizing any of its products.

This is probably due to the fact that the ISO 9000 certificate is issued for an organization's process and not 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 (RVC).

11.5.5 Summary of ISO 9001 Requirements

A summary of the main requirements of ISO 9001 as they relate of software development are as follows. Section numbers in brackets correspond to those in the standard itself:

Management responsibility (4.1)

- The management must have an effective quality policy.
- The responsibility and authority of all those whose work affects quality must be defined and documented.
- A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
- The effectiveness of the quality system must be periodically reviewed by audits.

Quality System (4.2)

A quality system must be maintained and documented.

Contract reviews (4.3)

Before entering into a contract, an organization must review the contract to ensure that it is understood, and that the organization has the necessary capability for carrying out its obligations.

Design control (4.4)

- The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
- Design inputs must be verified as adequate.
- Design must be verified.
- Design output must be of required quality.
- Design changes must be controlled.

Document control (4.5)

- There must be proper procedures for document approval, issue and removal.
- Document changes must be controlled. Thus, use of some configuration management tools is necessary.

Purchasing (4.6)

Purchased material, including bought-in software must be checked for conforming to requirements.

Purchaser supplied product (4.7)

Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product identification (4.8)

The product must be identifiable at all stages of the process. In software terms this means *configuration management*.

Process control (4.9)

- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and testing (4.10)

In software terms this requires effective testing, i.e. unit testing, integration testing and system testing. Test records must be maintained.

Inspection, measuring and test equipment(4.11)

If integration, measuring, and test equipment are used, they must be properly maintained and calibrated.

Inspection and test status (4.12)

The status of an item must be identified. In software terms this implies configuration management and release control.

Control of non-conforming product (4.13)

In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

Corrective action (4.14)

This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the system needs improvement.

Handling (4.15)

This clause deals with the storage, packing, and delivery of the software product.

Quality records (4.16)

Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

Quality audits (4.17)

Audits of the quality system must be carried out to ensure that it is effective.

Training (4.18)

Training needs must be identified and met.

Various ISO 900 requirements are largely common sense. Official guidance on the interpretation of ISO 9001 is inadequate at the present time, and taking expert advice is usually worthwhile.

11.5.6 Salient Features of ISO 9001 Requirements

In section 11.4.5, we pointed out the various requirements for the ISO 9001 certification. We can summarize the salient features all the the requirements as follows:

1. **Document control:** All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.

2. **Planning:** Proper plans should be prepared and then progress against these plans should be monitored.

3. **Review:** Important documents across all phases should be independently checked and reviewed for effectiveness and correctness.

4. **Testing:** The product should be tested against specification.

5. **Organizational aspects:** Several organizational aspects should be addressed, e.g. management reporting of the quality team.

11.5.7 ISO 9000-2000

ISO revised the quality standards in the year 2000 to fine tune the standards. The major changes include a mechanism for continuous process improvement. There is also an increased emphasis on the role of the top management, including establishing a measurable objectives for various roles and levels of the organization. The new standard recognizes that there can be many processes in an organization.

11.5.8 Shortcomings of ISO 9000 Certification

Even though ISO 9000 is widely being used for setting up an effective quality system in an organization, it suffers from several shortcomings. Some of these shortcomings of the ISO 9000 certification process are the following:

- ISO 9000 requires a software production process to be adhered to, but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.

- ISO 9000 certification process is not foolproof and no international accredition agency exists. Therefore, it is likely that variations in the norms of awarding certificates can exist among the different accredition agencies and also among the registrars.
- Organizations getting ISO 9000 certification often tend to downplay domain expertise and the ingenuity of the developers. These organizations start to believe that since a good process is in place, the development results are truly person-independent. That is, any developer is as effective as any other developer in performing any particular software development activity. In manufacturing industry, there is a clear link between process quality and product quality. Once a process is calibrated, it can be run again and again producing quality goods. Many areas of software development are so specialized that special expertise and experience in these areas (domain expertise) is required. Also, unlike in case of general product manufacturing, ingenuity and effectiveness of personal practices play an important part in determining the results produced by a developer. In other words, software development is a creative process and individual skills and experience are important.
- ISO 9000 does not automatically lead to continuous process improvement. In other words, it does not automatically lead to TQM.

11.6 SEI CAPABILITY MATURITY MODEL

SEI Capability Maturity Model (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. CMM is patterned after the pioneering work of Philip Crosby who published his maturity grid of five evolutionary stages in adopting quality practices in his book "Quality is Free" [Crosby, 79].

The Unites States Department of Defence (US DoD) is the largest buyer of software product. It often faced difficulties in vendor performances, and had to many times live with low quality products, late delivery and cost escalations. In this context, SEI CMM was originally developed to assist the US Department of Defence (DoD) in software acquisition. The rationale was to include the likely contractor performance as a factor in contract awards. Most of the major DoD contractors began CMM-based process improvement initiatives as they vied for DoD contracts. It was observed that the SEI CMM model helped organizations to improve the quality of the software they developed and therefore adoption of SEI CMM model had significant business benefits. Gradually, many commercial organizations began to adopt CMM as a framework for their own internal improvement initiatives.

In simple words, CMM is a reference model for apprising the software process maturity into different levels. This can be used to predict the most likely outcome to be expected from the next project that the organization undertakes. It must be remembered that SEI CMM can be used two ways—capability evaluation and software process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organization. Capability evaluation is administered by the contract awarding authority, and therefore the results would indicate the likely contractor performance if the contractor is awarded a work. On the other hand, software process assessment is used by an organization with the objective to improve its own process capability. Thus, the latter type of assessment is for purely internal use by a company.

CHAPTER 13

SOFTWARE MAINTENANCE

Many students and practising engineers have a preconceived bias against software maintenance work. The mention of the word maintenance brings up the image of a screw driver wielding mechanic with soiled hands holding onto a bagful of spare parts. It would be the objective of this chapter to clear up this misnomer, provide some intuitive understanding of the software maintenance projects, and to familiarize you with the latest techniques in software maintenance.

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use. On the other hand, software products do not need maintenance on this count, but need maintenance to correct errors, enhance features, port to new platforms, etc.

In Section 13.1, we examine some general issues concerning maintenance projects. In Section 13.2, we discuss some basic ideas about software reverse engineering. In Section 13.3, we discuss two software maintenance process models which attempt to systematize the software development effort, and finally we discuss some concepts involved in cost estimation of maintenance efforts.

13.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE

In this section, we first classify the different maintenance efforts into a few classes. Next, we discuss some general characteristics of the maintenance projects. We also discuss some special problems associated with maintenance projects.

Software maintenance is becoming an important activity of a large number of organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts.

13.1.1 Types of Software Maintenance

Software maintenance can be required for three main reasons as follows:

1. Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

2. Adaptive: A software 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 interface with new hardware or software.

3. Perfective: A software 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.

13.1.2 Characteristics of Software Evolution

Lehman and Belady have studied the characteristics of evolution of several software products [1980]. They have expressed their observations in the form of laws. Their important laws are presented in the following. But a word of caution here is that these are generalizations and may not be applicable to specific cases and also most of these observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

Lehman's first law

A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs, and therefore, tend to incur higher maintenance efforts. This law clearly shows that every product irrespective of how well-designed must undergo maintenance. In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law

The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that when you add a function during maintenance, you build on top of an existing program, often in a way that the existing program was not intended to support. If you do not redesign the system, the additions will be more complex that they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law

Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore, this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

13.1.3 Special Problems Associated with Software Maintenance

Software maintenance work currently is typically much more expensive than what it should be and takes more time than required. The reasons for this situation are the following:

Software maintenance work in organizations is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand some one else's work, and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products. Though the word legacy implies "aged" software, but there is no agreement on what exactly is a legacy system. It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

13.2 SOFTWARE REVERSE ENGINEERING

Software 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 is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

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 way to carry out these cosmetic changes is shown schematically in Figure 13.1. A program can be reformatted using any of the several available prettyprinter programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important because we had seen in Chapter 9 that meaningful variable names is the most helpful code documentation. 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.

After the cosmetic changes have been carried out on a legacy software, the proces of extracting the code, design, and the requirements specification can begin. These activities are

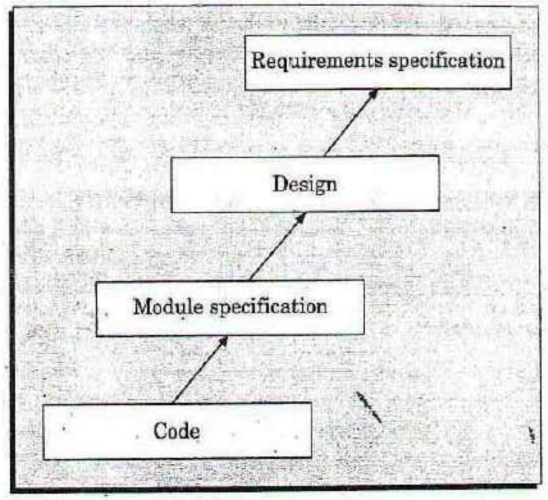

Figure 13.1: A process model for reverse engineering.

schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and 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.

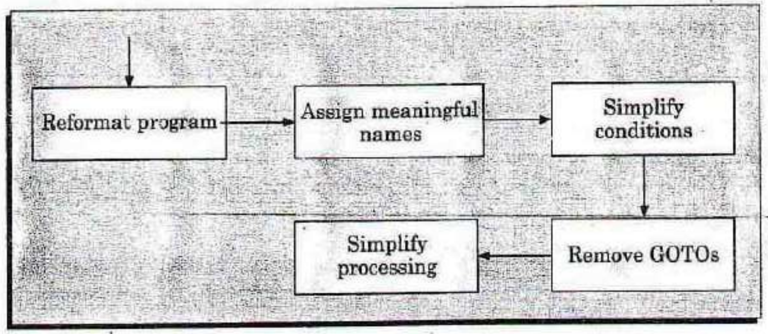

Figure 13.2: Cosmetic changes carried out before reverse engineering.

13.3 SOFTWARE MAINTENANCE PROCESS MODELS

Before discussing process models for software maintenance, we need to analyze various activities involved in a typical software maintenance project. The activities involved in a software maintenance project are not unique and depend on several factors such as the extent of modification to the product required, (the resources available to the maintenance team, the conditions of the existing product (e.g. how structured it is, how well documented it is, etc.), the expected project risks, etc. When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately 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 software 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 (activities required) 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 documents later. This maintenance process is graphically presented in Figure 13.3.

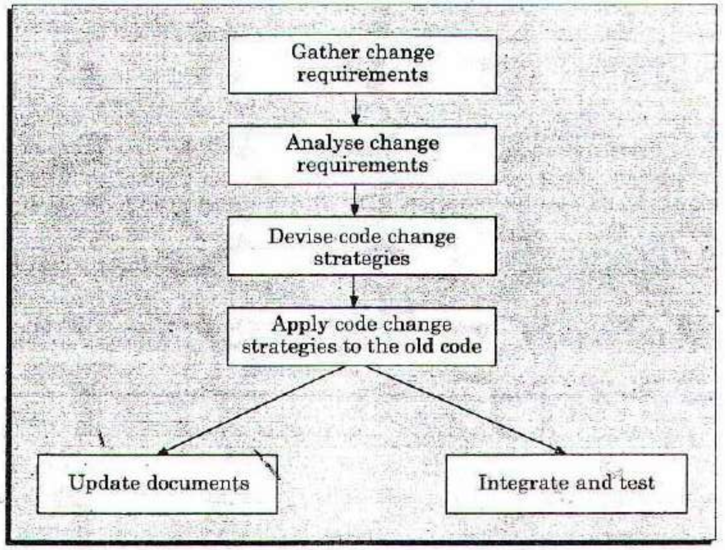
Gather change requirements
Analyse change requirements
Devise code change strategies
Apply code change strategies to the old code
Update documents
Integrate and test

Figure 13.3: 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 maintenance team as they 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. Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

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 **software reengineering**. This process model is depicted in Figure 13.4. The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed (abstracted) to extract the module

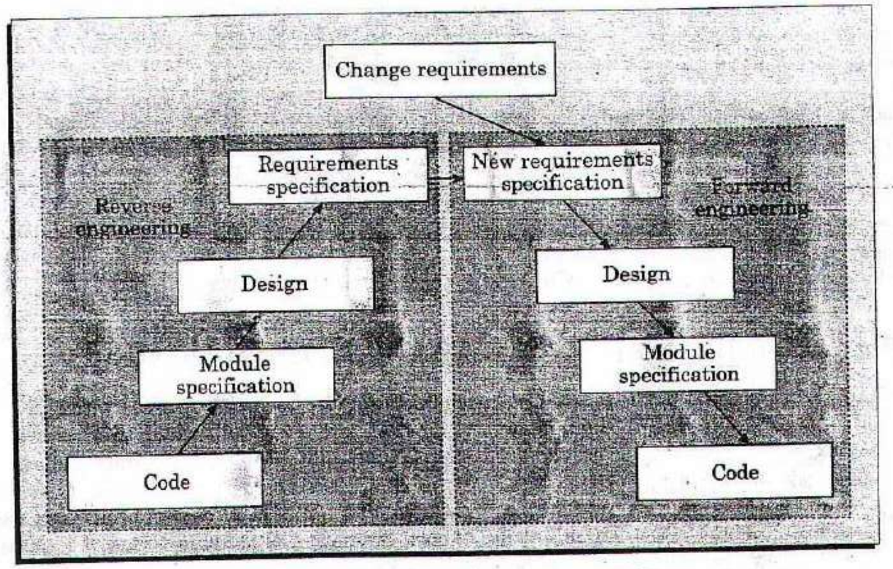

Figure 13.4: Maintenance process model 2.

specifications. The module specifications are then analyzed to produce the design. The design is analyzed (abstracted) 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 a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15% (see Figure 13.5). Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

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

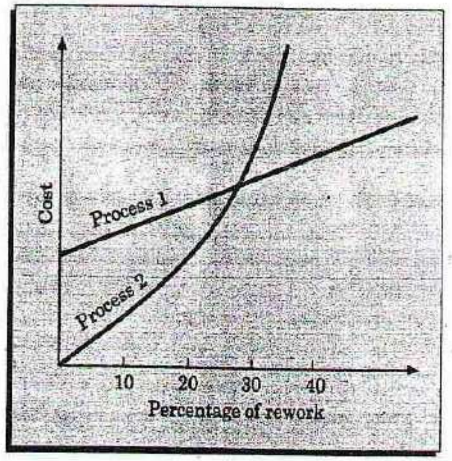

Figure 13.5: Empirical estimation of maintenance cost versus percentage rework.

13.4 ESTIMATION OF MAINTENANCE COST

We had earlier pointed out that maintenance efforts require about 60% of the total life cycle cost for a typical software 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, 1981 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 quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

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

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 code deleted.

The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost, i.e.

$$\text{Maintenance cost} = ACT \times \text{Development cost}$$

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

SUMMARY

- In this chapter, we discussed some fundamental concepts associated with software maintenance activities.
- Maintenance is the most expensive phase of the software life cycle and therefore it is usually cost-effective to invest in time and effort while developing the product and to emphasize on maintainability of the product to reduce the maintenance costs.
- We discussed the activities in reverse engineering and then discussed two maintenance process models. We also discussed the applicability of these two process models to maintenance projects.
- We highlighted the salient points in costing maintenance projects.

EXERCISES

1. What are the different types of maintenance that a software product might need? Why are these maintenance required?.
2. Explain why every software system must undergo maintenance or progressively become less useful.
3. Discuss the process models for software maintenance and indicate how you would select an appropriate maintenance model for a maintenance project at hand.
4. State whether the following statements are **TRUE** or **FALSE**. Give reasons for your answer.
   (a) Legacy software products are those products which have been developed long time back.
   (b) Corrective maintenance is the type of maintenance that is most frequently carried out on a typical software product.
5. What do you mean by the term software reverse engineering? Why is it required? Explain the different activites undertaken during reverse engineering.
6. What do you mean by the term software reengineering? Why is it required?
7. If a software product costed Rs. 10,000,000/- for development, compute the annual maintenance cost given that every year approximately 5% of the code needs modification. Identify the factors which render the maintenance cost estimation inaccurate.
8. What is a legacy software product? Explain the problems one would encounter while maintaining a legacy product.

CHAPTER 14

SOFTWARE REUSE

Software products are expensive. Therefore, software project managers are always worried about the high cost of software development, and are desperately looking for way-outs to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality. A reuse approach that is of late gaining prominence is component-based development. Component-based software development is different from the traditional software development in that software is developed by assembling software from off-the-shelf components.

Software development with reuse is very similar to a modern hardware engineer building an electronic circuit by using standard types of ICs and other components. In this chapter, we will review the state of art in software reuse.

14.1 WHAT CAN BE REUSED?

Before discussing the details of reuse techniques, it is important to deliberate about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

- Requirements specification
- Design
- Code
- Test cases
- Knowledge

Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts, reuse of knowledge occurs automatically without any conscious effort in this direction. However, two major difficulties with unplanned reuse of knowledge is that a developer experienced in one type of product might be included in a team developing a different type of software. Also, it is difficult to remember the details of the potentially reusable development knowledge. A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

14.2 WHY ALMOST NO REUSE SO FAR?

A common scenario in many software development industries is explained further.

Engineers working in software development organizations often have a feeling that the current system that they are developing is similar to the last few systems built. However, no attention is paid on how not to duplicate what can be reused from previously developed systems. Everything is being built from the scratch. The current system falls behind schedule and no one has time to figure out how the similarity between the current system and the systems developed in the past can be exploited.

Even those organizations which embark on a reuse program, in spite of the above difficulty, face other problems. Creation of components that are reusable in different applications is a difficult problem. It is very difficult to anticipate the exact components that can be reused across different applications. But, even when the reusable components are carefully created and made available for reuse, programmers prefer to create their own, because the available components are difficult to understand and adapt to the new applications.

In this context, the following observation is significant

The routines of mathematical libraries are being reused very successfully by almost every programmer. No one in their mind would think of writing a routine to compute sine or cosine. Let us investigate why reuse of commonly used mathematical functions is so easy. Several interesting aspects emerge. Cosine means the same to all. Everyone has clear ideas about what kind of argument should cosine take, the type of processing to be carried out and the results returned. Secondly, mathematical libraries have a small interface. For example, cosine requires only one parameter. Also, the data formats of the parameters are standardized. These are some fundamental issues which would remain valid for all our subsequent discussions on reuse. In the following section, we discuss the issues that must be addressed while starting any reuse program in an organization.

14.3 BASIC ISSUES IN ANY REUSE PROGRAM

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

- Component creation
- Component indexing and storing
- Component search
- Component understanding
- Component adaptation
- Repository maintenance

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. In Section 14.4, we discuss domain analysis as a promising technique which can be used to create reusable components.

Component 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.

Component searching

The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

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.

Component adaptation

Often, the components may need adaptation before they can be reused, since a selected component may 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.

Repository maintenance

A component repository once is created requires continuous maintenance. New components, as and when created have to be entered 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.

14.4 A REUSE APPROACH

A promising approach that is being adopted by many organizations is to introduce a building block approach into the software 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. It must be clearly understood that an issue crucial to every reuse effort is the identification of reusable components. Domain analysis is a promising approach to identify reusable components. In the following, we discuss the domain analysis approach to create reusable components.

14.4.1 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 software product. A reuse domain is a shared understanding of some community, characterized by concepts, techniques, and terminologies that show some coherence. Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.

Just to become familiar with the vocabulary of a domain requires months of interaction with the experts. Often, one needs to be familiar with a network of related domains for successfully carrying out domain analysis. Domain analysis identifies the objects, operations, and the relationships among them. For example, consider the airline reservation system, the reusable objects can be seats, flights, airports, crew, meal orders, etc. The reusable operations can be scheduling a flight, reserving a seat, assigning crew to flights, etc. We can see that the domain analysis generalizes the application domain. A domain model transcends specific applications. The common characteristics or the similarities between systems are generalized.

During domain analysis, a specific community of software developers get together to discuss community-wide solutions. Analysis of the application domain is required to identify the reusable components. The actual construction of the reusable components for a domain is called **domain engineering**.

Evolution of a reuse domain

The ultimate results of domain analysis is development of problem-oriented languages. The problem-oriented languages are also known as **application generators**. These application generators, once developed form application development standards. The domains slowly develop. As a domain develops, we may distinguish the various stages it undergoes:

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

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

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

4. Stage 4: The domain has been fully explored. The software development for the domain can be largely be 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**.

14.4.2 Component Classification

Components need to be properly classified in order to develop an effective indexing and storage scheme. We have already remarked that hardware reuse has been very successful. If we look at the classification of hardware components for clue, then we can observe that hardware components are classified using a multilevel hierarchy. 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.