# UNIT-II

# CHAPTER-I

# SOFTWARE PROJECT MANAGEMENT

Software project management is a very vast topic. In fact, a full semester teaching can be conducted on effective techniques for software project management.

The main goal of software project management is to enable a group of developers to work effectively towards the successful completion of a project.

## SOFTWARE PROJECT MANAGEMENT COMPLEXITIES

- **Invisibility:** Invisibility of software makes it difficult to assess the progress of a project and is a major cause for the complexity of managing a software project.
- **Changeability:** Frequent changes to the requirements and the invisibility of software are possibly the two major factors making software project management a complex task.
- **Complexity:** Even a moderate sized software has millions of parts (functions) that interact with each other in many ways—data coupling, serial and concurrent runs, state transitions, control dependency, file sharing, etc.
- **Uniqueness:** Every software project is usually associated with many unique features or situations. This makes every project much different from the others. This is unlike projects in other domains, such as car manufacturing or steel manufacturing where the projects are more predictable. Due to the uniqueness of the software projects, a project manager in the course of a project faces many issues that are quite unlike the ones he/she might have encountered in the past.
- **Exactness of the solution:** Mechanical components such as nuts and bolts typically work satisfactorily as long as they are within a tolerance of 1 per cent or so of their specified sizes. However, the parameters of a function call in a program are required to be in complete conformity with the function definition.
- **Team-oriented and intellect-intensive work:** Software development projects are akin to research projects in the sense that they both involve team-oriented, intellect-intensive work. In contrast, projects in many domains are labor-intensive and each member works in a high degree of autonomy. Examples of such projects are planting rice, laying roads, assembly line manufacturing, constructing a multistoried building, etc.

## RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

### Job Responsibilities for Managing Software Projects

A software project manager takes the overall responsibility of steering a project to success. This surely is a very hazy job description. In fact, it is very difficult to objectively describe the precise job responsibilities of a project manager. The job responsibilities of a project manager range from invisible activities like building up of team morale to highly visible customer presentations.

We can broadly classify a project manager's varied responsibilities into the following two major categories:

- Project planning, and
- Project monitoring and control.

**Project planning:** Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase.

The initial project plans are revised from time to time as the project progresses andmore project data become available.

**Project monitoring and control:** Project monitoring and control activities are undertaken once the development activities start. As the project gets underway, the details of the project that were unclear earlier at the start of the project emerge and situations that were not visualized earlier arise. While carrying out project monitoring and control activities, a project manager usually needs to change the plan to cope up with specific situations at hand.

## Skills Necessary for Managing Software Projects

A theoretical knowledge of various project management techniques is certainly important to become a successful project manager. However, a purely theoretical knowledge of various project management techniques would hardly make one a successful project manager. Effective software project management calls for good qualitative judgment and decision taking 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 to get work done. Some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. Never the less, the importance of a sound knowledge of the prevalent project management techniques cannot be overemphasized.

- Three skills that are most critical to successful project management
- are the following:
- Knowledge of project management techniques.
- Decision taking capabilities
- Previous experience in managing similar projects

## PROJECT PLANNING

Project managers undertake project planning. Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale.

It can even cause project failure. For this reason, project planning is undertaken by the project managers with utmost care and attention. However, for effective project planning, in addition to a thorough knowledge of the various estimation techniques, past experience is crucial.

During project planning, the project manager performs the following activities. Note that we have given only a very brief description of the activities.

- **Estimation:** The following project attributes are estimated.

  Cost: How much is it going to cost to develop the software product?

  Duration: How long is it going to take to develop the product?

  Effort: How much effort would be necessary to develop the product?

The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

- **Scheduling:** After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.
- **Staffing:** Staff organization and staffing plans are made.
- **Risk management:** This includes risk identification, analysis, and abatement planning.
- **Miscellaneous plans:** This includes making several other plans such as quality assurance plan, and configuration management plan, etc.
- Figure 3.1 shows the order in which the planning activities are undertaken. Observe that size estimation is the first activity that a project manager undertakes during project planning.
- As can be seen from Figure 3.1, based on the size estimation, the effort required to complete a 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 price negotiations with the customer is carried out. Other planning activities such as staffing, scheduling etc. are undertaken based on the effort and duration estimates made. In Section 3.7, we shall discuss a popular technique for estimating the project parameters. Subsequently, we shall discuss the staffing and scheduling issues.
- Size is the most fundamental parameter, based on which all other estimations and project plans are made.

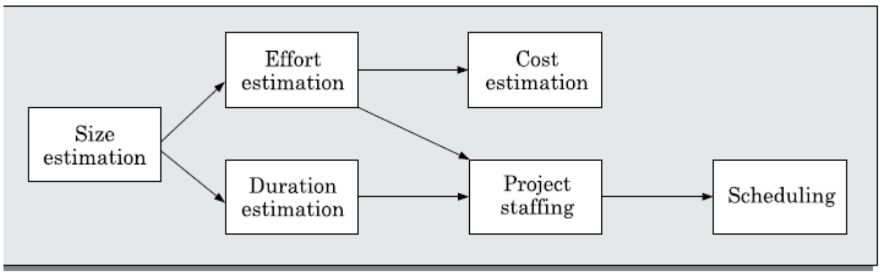


FIGURE 3.1 Precedence ordering among planning activities.

## Sliding Window Planning

- It is usually very difficult to make accurate plans for large projects at project initiation. A part of the difficulty arises from the fact that large projects may take several years to complete.
- As a result, during the span of the project, the project parameters, scope of the project, project staff, etc., often change drastically resulting in the initial plans going haywire. In order to overcome this problem, sometimes project managers undertake project planning over several stages.
- That is, after the initial project plans have been made, these are revised at frequent intervals. Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning known as *sliding window planning*.
- At the start of a project, the project manager has incomplete knowledge about the nitty-gritty of the project. His information base gradually improves as the project progresses through different development phases. The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear. The project parameters are re-estimated periodically as understanding grows and also a periodically as project parameters change.

## The SPMP Document of Project Planning

- Once project planning is complete, project managers document their plans in a software project management plan (SPMP) document. Listed below are the different items that the SPMP document should discuss. This list can be used as a possible organization of the SPMP document.

**Organization of the software project management plan (SPMP) document**

- **1. Introduction**

  (a) Objectives

  (b) Major Functions

  (c) Performance Issues

  (d) Management and Technical Constraints

- **2. Project estimates**

  (a) Historical Data Used

  (b) Estimation Techniques Used

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

- **3. Schedule**

  (a) Work Breakdown Structure

(b) Task Network Representation

(c) Gantt Chart Representation

(d) PERT Chart Representation

- **4. Project resources**

  (a) People

  (b) Hardware and Software

  (c) Special Resources

- **5. Staff organisation**

  (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**

  (a) Metrics to be tracked

  (b) Tracking plan**6**

  (c) 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

## METRICS FOR PROJECT SIZE ESTIMATION

- 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 measure size—lines of code (LOC) and function point (FP). Each of these metrics has its own advantages and disadvantages.
- Based on their relative advantages, one metric may be more appropriate than the other in a particular situation.

**Lines of Code (LOC)**

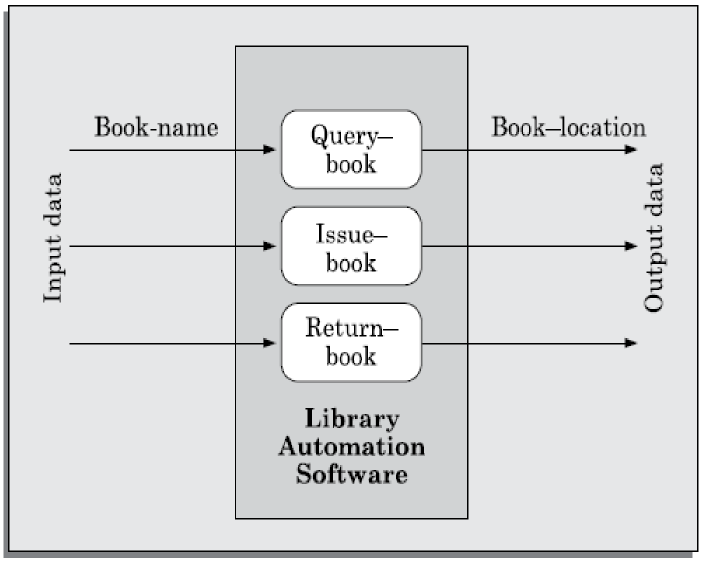
- LOC is possibly the simplest among all metrics available to measure project size.
- Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.
- Determining the LOC count at the end of a project is very simple. However, accurate estimation of LOC count at the beginning of a project is a very difficult task. One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and
- **LOC is a measure of coding activity alone.** The implicit assumption made by the LOC metric that the overall product development effort is solely determined from the coding effort alone is flawed.
- **LOC count depends on the choice of specific instructions:** LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers.
- Even for the same programming problem, different programmers might come up with programs having very different LOC counts. This situation does not improve, even if language tokens are counted instead of lines of cod
- **LOC measure correlates poorly with the quality and efficiency of the code:** Larger code size does not necessarily imply better quality of code or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms.
- **LOC metric penalizes use of higher-level programming languages and code reuse:** A 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, and in turn, would indicate lower effort!
- **LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities:** Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic.
- **It is very difficult to accurately estimate LOC of the final program from problem specification:** As already discussed, at the project initiation time, it is a very difficult

task to accurately estimate the number of lines of code (LOC) that the program would have after development. The LOC count can accurately be computed only after the code has fully been developed. Since project planning is carried out even before any development activity starts, the LOC metric is of little use to the project managers during project planning.

**Function Point (FP) Metric**

- Function point metric was proposed by Albrecht and Gaffney in 1983. This metric overcomes many of the shortcomings of the LOC metric. Since its inception, function point metric has steadily gained popularity. Function point metric has several advantages over LOC metric.
- One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself.
- Using the LOC metric, on the other hand, the size can accurately be determined only after the code has been fully written.
- The conceptual idea behind the function point metric is the following. The size of a software product is directly dependent on the number of different high-level functions or features it supports. This assumption is reasonable, since each feature would take additional effort to implement.
- Though each feature takes some effort to develop, different features may take very different amounts of efforts to develop. For example, in a banking software, a function to display a help message may be much easier to develop compared to say the function that carries out the actual banking transactions. Therefore, just determining the number of functions to be supported (with adjustments for number of fi les and interfaces) may not yield very accurate results.
- This has been considered by the function point metric by counting the number of input and output data items and the number of files accessed by the function. The implicit assumption made is that the more the number of data items that a function reads from the user and outputs and the more the number of files accessed, the higher is the complexity of the function.
- Now let us analyze why this assumption must be intuitively correct. Each feature when invoked typically reads some input data and then transforms those to the required 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 in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding
- input data. It can therefore be argued that the computation of the number of input and
- output data items would give a more accurate indication of the code size compared to

  simply counting the number of high-level functions supported by the system.



RE 3.2 System function as a mapping of input data to output data

**Function point (FP) metric computation**

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

- Step 1: Compute the unadjusted function point (UFP) using a heuristic expression.
- Step 2: Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.
- Step 3: Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort. We discuss these three steps in more detail in the following.

**Step 1: UFP computation**

- The *unadjusted function points* (UFP) are computed as the weighted sum of five characteristics of a product as shown in the following expression. The weights associated with the five characteristics were determined empirically by Albrecht through data gathered from manyprojects.
- UFP = (Number of inputs) *4 + (Number of outputs) *5 + (Number of inquiries) *4

+ (Number of files) *10 + (Number of interfaces) *10 (3.1)

The meanings of the different parameters of Eq. (3.1) are as follows:

- 1. **Number of inputs:** Each data item input by the user is counted. However, it should be noted that data inputs are considered different from user inquiries. Inquiries are user commands such as print-account-balance that require no data values to be input by the user. Inquiries are counted separately (see the third point below).

  It needs to be further noted that individual data items input by the user are not simply added up to compute the number of inputs, but related inputs are grouped and considered as a single input. For example, while entering the data concerning an employee to an employee pay roll 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 describe a single employee.

- 2. **Number of outputs:** The outputs considered include 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 just a single output.
- 3. **Number of inquiries:** An inquiry is a user command (without any data input) and only requires some actions to be performed by the system. Thus, the total number of inquiries is essentially the number of distinct interactive queries (without data input) which can be made by the users. Examples of such inquiries are print account balance, print all student grades, display rank holders' names, etc.
- 4. **Number of files:** The *files* referred to here are logical files. A logical file represents a group of logically related data. Logical files include data structures as well as physical files.
- 5. **Number of interfaces:** Here the interfaces denote the different mechanisms that are used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.

**Step 2: Refine parameters**

- UFP computed at the end of step 1 is a gross indicator of the problem size. This UFP needs to be refined by taking into account various peculiarities of the project. This is possible, since each parameter (input, output, etc.) has been implicitly assumed to be of average complexity. However, this is rarely true. For example, some input values may be extremely complex, some very simple, etc. In order to take this issue into account, UFP is refined by taking into account the complexities of the parameters of UFP computation (Eq. 3.1). The complexity of each parameter is graded into three broad categories—simple, average, or complex.
- The weights for the different parameters are determined based on the numerical values shown in Table 3.1. Based on these weights of the parameters, the parameter values in the UFP are refined. For example, rather than each input being computed as four FPs, very simple inputs are computed as three FPs and very complex inputs as six FPs.

**Step 3: Refine UFP based on complexity of the overall project**

- In the final step, several factors that can impact the overall project size are considered to refine the UFP computed in step 2. Examples of such project parameters that can influence the project sizes include high transaction rates, response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort.
- The list of these parameters has been shown in Table 3.2. Each of these 14 parameters 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). 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 the corresponding project parameters on the development effort. TCF is computed as (0.65 + 0.01 × DI).

As DIcan vary from 0 to 84, TCF can vary from 0.65 to 1.49. Finally, FP is given as the product of UFP and TCF. That is, FP = UFP × TCF.

| 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 |

**TABLE 3.1** Refinement of Function Point Entities

**TABLE 3.2** Function Point Relative Complexity Adjustment Factors

| |
|---|
| Requirement for reliable backup and recovery |
| Requirement for data communication |
| Extent of distributed processing |
| Performance requirements |
| Expected operational environment |
| Extent of online data entries |
| Extent of multi-screen or multi-operation online data input |
| Extent of online updating of master files |
| Extent of complex inputs, outputs, online queries and files |
| Extent of complex data processing |
| Extent that currently developed code can be designed for reuse |
| Extent of conversion and installation included in the design |
| Extent of multiple installations in an organisation and variety of customer organisations |
| Extent of change and focus on ease of use |

**Feature point metric shortcomings:** A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a function. That is, the function point metric implicitly assumes that the effort required to design and develop any two different functionalities of the system is the same.

But we know that this is highly unlikely to be 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. FP only considers the number of functions that the system supports, without distinguishing the difficulty levels of developing the various functionalities. To overcome this problem, an extension to 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 higher the complexity of a function, the greater the effort required to develop it— therefore, it should have larger size compared to a simpler function.

**Critical comments on the function point and feature point metrics**

Proponents of function point and feature point metrics claim that these two metrics are language-independent and can be easily computed from the SRS document during project planning stage itself. On the other hand, 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 groups input and output data items into logically related groups can be very subjective. For example, consider that certain functionality requires the employee's name and employee address to be input. It is possible that one can consider both these items as a single unit of data, since after all, these describe a single employee.

It is also possible for someone else to consider an employee's address as a single unit of input data and name as another. Such ambiguities leave sufficient scope for debate and keep open the possibility for different project managers to arrive at different function point measures for essentially the same problem.

**PROBLEM 3.1** Determine the function point measure of the size of the following supermarket software. A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique *customer number* (CN) by the computer. Based on the generated CN, a clerk manually prepares a customer identity card after getting the market manager's signature on it. A customer can present his customer identity card to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded `10,000. The entries against the CN are reset on the last day of every year after the prize winners' lists are generated. Assume that various project characteristics determining the complexity of software development to be average.

***Solution:***

**Step 1:** From an examination of the problem description, we find that there are two inputs,

three outputs, two files, and no interfaces. Two files would be required, one for storing the customer details and another for storing the daily purchase records. Now, using equation 3.1, we get:

UFP = 2 × 4 + 3 × 5 + 1 × 4 + 10 × 2 + 0 × 10 = 47

**Step 2:** All the parameters are of moderate complexity, except the output parameter of customer registration, in which the only output is the CN value. Consequently, the complexity of the output parameter of the customer registration function can be categorized as simple. By consulting Table 3.1, we find that the value for simple output is given to be 4. The UFP can be refined as follows:

UFP = 3 × 4 + 2 × 5 + 1 × 4 + 10 × 2 + 0 × 10 = 46

Therefore, the UFP will be 46.

**Step 3:** Since the complexity adjustment factors have average values, therefore the total degrees of influence would be: DI = 14 × 4 = 56

$TCF = 0.65 + 0.01 + 56 = 1.21$

Therefore, the adjusted FP = 46 × 1.21 = 55.66

## PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is an important project planning activity. The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost. Accurate estimation of these parameters is important, since these not only help in quoting an appropriate project cost to the customer, but also form the basis for resource planning and scheduling.

A large number of estimation techniques have been proposed by researchers. These can broadly be classified into three main categories:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

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

- **Empirical Estimation Techniques**

  Empirical estimation techniques are essentially 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 and subjective decisions, over the years, the different activities involved in estimation have been formalized to a large extent. We shall discuss two such formalizations of the basic empirical estimation techniques known as expert judgement and the Delphi techniques in Sections

- **Heuristic Techniques**

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

  Single variable estimation models assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterized by an expression of the following form:

- Estimated Parameter = $c1 \times ed1$

- In the above expression, $e$ represents a characteristic of the software that 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., $c1$ and $d1$ are constants. The values of the constants $c1$ and $d1$ are usually determined using data collected from past projects (historical data). The COCOMO model discussed in Section 3.7.1, is an example of a single variable cost estimation model.

- A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:

- Estimated Resource = $c1 \times pd1 + c2 \times pd2 + L$

- where, $p1$, $p2$, ... are the basic (independent) characteristics of the software already estimated, and $c1$, $c2$, $d1$, $d2$, .... 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 different sets of constants $c1$, $d1$, $c2$, $d2$, .... Values of these constants are usually determined from an analysis of historical data. The intermediate COCOMO model discussed in Section 3.7.2 can be considered to be an example of a multivariable estimation model

**Analytical Estimation Techniques**

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project. Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis. As an example of an analytical technique, we shall discuss the Halstead's software science in Section 3.8. We shall see that starting with a few simple assumptions, Halstead's software science derives some interesting results. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques as far as estimating software maintenance efforts is concerned.

## EMPIRICAL ESTIMATION TECHNIQUES

- We have already pointed out that empirical estimation techniques have, over the years, been formalized to a certain extent. Yet, these are still essentially euphemisms for pure guess work. These techniques are easy to use and give reasonably accurate estimates. Two popular empirical estimation techniques are—Expert judgement and Delphi estimation techniques. We discuss these two techniques in the following subsection.
- **Expert Judgement**

  Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about 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 then individual modules to arrive at the overall estimate. However, this technique suffers from several shortcomings. The outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant 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. Due to these factors, the size estimation arrived at by the judgement of a single expert may be far from being accurate.
- A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues 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 is minimized when the estimation is done by a group of experts.
- However, the estimate made by a group of experts may still exhibit bias. For example, on certain issues the entire group of experts may be biased due to reasons such as those arising out of political or social considerations. Another important shortcoming of the expert judgement technique is that the decision made by a group may be dominated by overly assertive members.

### 3.6.2 Delphi Cost Estimation

- Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the *software requirements specification* (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The coordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process.

- The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate. The Delphi estimation, though consumes more time and effort, overcomes an important shortcoming of the expert judgement technique in that the results cannot unjustly be influenced by overly assertive and senior members.

## COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

Constructive Cost estimation Model (COCOMO) was proposed by Boehm [1981]. COCOMO prescribes a three-stage process for project estimation. In the first stage, an initial estimate is arrived at. Over the next two stages, the initial estimate is refined to arrive at a more accurate estimate. COCOMO uses both single and multivariable estimation models at different stages of estimation.

The three stages of COCOMO estimation technique are—basic COCOMO, intermediate COCOMO, and complete COCOMO. We discuss these three stages of estimation in the following subsection.

### Basic COCOMO Model

- Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity—*organic, semidetached, and embedded*. Based on the category of a software development project, he gave different sets of formulas to estimate the effort and duration from the size estimate.

Three basic classes of software development projects

In order to classify a project into the identified categories, Boehm requires us to consider not only the characteristics of the product but also those of the development team and development environment. Roughly speaking, the three product development classes correspond to development of application, utility and system software. Normally, data= processing programs2 are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and programming complexities also arise out of the requirement for meeting timing constraints and concurrent processing of tasks.

- Brooks [1975] states that utility programs are roughly three times as difficult to write as application programs and system programs are roughly three times as difficult as utility programs. Thus, according to Brooks, the relative levels of product development complexity for the three categories (application, utility and system programs) of products are 1:3:9.

- Boehm's [1981] definitions of organic, semidetached, and embedded software are elaborated as follows:

**Organic:** We can classify a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

**Semidetached:** A development project can be classified to be of semidetached type, if the development team consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

**Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist. development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

- Observe that in deciding the category of the development project, in addition to considering the characteristics of the product being developed, we need to consider the characteristics of the team members. Thus, a simple data processing program may be classified as semidetached, if the team members are inexperienced in the development of similar products.
- For the three product categories, Boehm provided different sets of constant values for the coefficients for the two basic expressions to predict the effort (in units of person-months) and development time from the size estimation given in kilo lines of source code (KLSC). But how much effort is one person-month?
- Person-month (PM) is a popular unit for effort measurement. It should be carefully noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither does it imply that 1 person should be employed for 100 months to complete the project.
- The effort estimation simply denotes the area under the person-month curve (see Figure 3.3) for the project. The plot in Figure 3.3 shows that different number of personnel may work at different points in the project development. The number of personnel working on the project usually increases or decreases by an integral number resulting in the sharp edges in the plot. We shall elaborate in Section 3.9 how the exact number of persons to work at any time on the product development can be determined from the effort and duration estimates

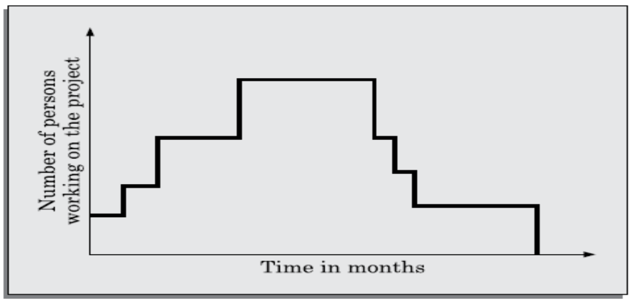


FIGURE 3.3 Person-month curve.

**General form of the COCOMO expressions**

- The **basic COCOMO model** is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms:
- Effort = $a1 \times (KLOC)a2$ PM
- Tdev = $b1 \times (Effort)b2$ months
- where,  KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- $a1, a2, b1, b2$ are constants for each category of software product.
- Tdev is the estimated time to develop the software, expressed in months.
-  Effort is the total effort required to develop the software product, expressed in person-months (PMs).
- According to Boehm, every line of source text should be calculated as one LOC n irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say *n* lines), it is considered to be *n* LOC. The values of $a1, a2, b1, b2$ for different categories of products as given by Boehm [1981] are summarized below.
- He derived these values by examining historical data collected from a large number of n actual projects.

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

- Organic: Effort = 2.4(KLOC)1.05 PM
- Semi-detached: Effort = 3.0(KLOC)1.12 PM
- Embedded: Effort = 3.6(KLOC)1.20 PM
- **Estimation of development time:** For the three classes of software products, the formulas for estimating the development time based on the effort are given below:
- Organic: Tdev = 2.5(Effort)0.38 Months

- Semi-detached: Tdev = 2.5(Effort)0.35 Months
- Embedded: Tdev = 2.5(Effort)0.32 Months We can gain some insight into the basic COCOMO model, if we plot the estimated effort and duration values for different software sizes. Figure 3.4 shows the plots of estimated effort versus product size for different categories of software products.

**Observations from the effort-size plot**

- From Figure 3.4, we can observe that the effort is somewhat super linear (that is, slope of the curve>1) in the size of the software product. This is because the exponent in the effort expression is more than 1. Thus, the effort required to develop a product increases rapidly with project size. The reason for this is that COCOMO assumes that projects are carefully designed and developed by using software engineering principles

**Observations from the development time—size plot**

- The development time versus the product size in KLOC is plotted in Figure 3.5. From Figure 3.5, we can observe the following:
- The development time is a sublinear function of the size of the product. That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately. For example, to develop a product twice as large as a product of size 100KLOC, the increase in duration may only be 20 per cent. It may appear surprising that the duration curve does not increase super linearly—one would normally expect the curves to behave similar to those in the effort-size plots. This apparent anomaly can be explained by the fact that COCOMO assumes that a project development is carried out not by a single person but by a team of developers.

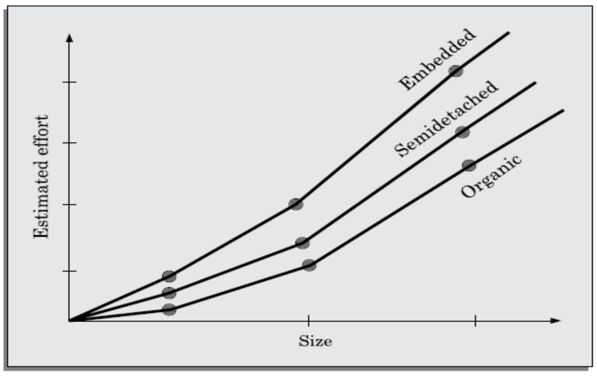


FIGURE 3.4 Effort versus product size.

- From Figure 3.5 we can observe that for a project of any given size, the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semi-detached, or embedded type. (Please verify this using the basic COCOMO formulas discussed in this section). However, according to the COCOMO formulas, embedded programs require much higher effort than either application or utility programs. We can interpret it to mean that there is more scope for parallel activities for system programs than those in utility or application programs.

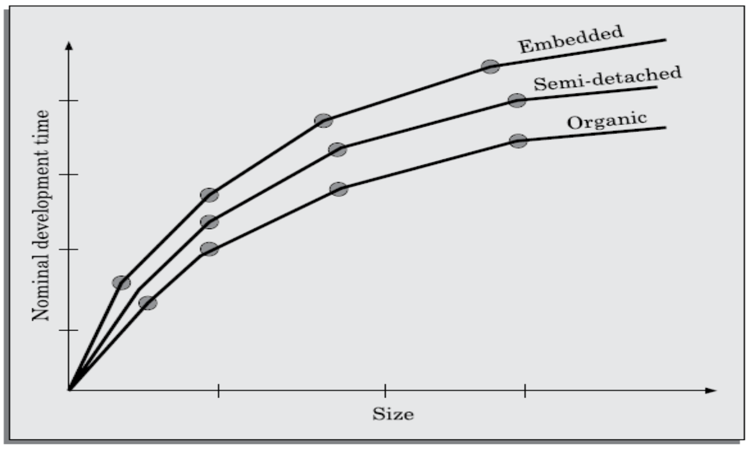


FIGURE 3.5 Development time versus size.

- **Cost estimation**

  From the effort estimation, project cost can be obtained by multiplying the estimated effort (in man-month) by the manpower cost per month. Implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. However, in addition to manpower cost, a project would incur several other types of costs which we shall refer to as the overhead costs. The overhead costs would include the costs due to hardware and software required for the project and the company overheads for administration, office space, electricity, etc. Depending on the expected values of the overhead costs, the project manager has to suitably scale up the cost arrived by using the COCOMO formula.

- **Implications of effort and duration estimate**

  The effort and duration values computed by COCOMO are the values for completing the work in the shortest time without unduly increasing manpower cost.

- **Staff-size estimation**

  Given the estimations for the project development effort and the nominal development time, can the required staffing level be determined by a simple division of the effort estimation by the duration estimation? The answer is "No". It will be a perfect recipe for project delays and cost overshoot. We examine the staffing problem in more detail in Section 3.9. From the discussions in Section 3.9, it would become clear that the simple division approach to obtain the staff size would be highly improper.

- **Development environment:** Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

## RISK MANAGEMENT

- Every project is susceptible to a large number of risks. Without effective management of the risks, even the most meticulously planned project may go haywire. We need to distinguish between a risk which might occur from the risks that have already become real and are currently being faced by a project.
- If a risk becomes real, 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 beforehand to contain each risk. In this context, risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real. Risk management consists of three essential activities—
- Risk identification,
- Risk assessment, and
- Risk mitigation

**Risk Management Approaches**
Risk management approaches can broadly be classified into reactive and proactive approaches. The later approach is much more effective in risk handling and therefore used wherever possible. In the following, we briefly discuss these two approaches

**Reactive approaches**
Reactive approaches take no action until an unfavorable event occurs. Once an unfavorable event occurs, these approaches try to contain the adverse effects associated with the risk and take steps to prevent future occurrence of the same risk events. An example of such a risk management strategy can be the following. Consider a project in which the server hosting the project data crashes. Once this risk event has occurred, the team members may put best effort to recover the data and also initiate the practice of taking regular backups, so that in future such a risk event does not recur. It is similar to calling the emergency firefighting service once a fire has been noticed, and then installing firefighting equipment in all rooms of the building to be able to instantly handle fire the next time it is noticed. It can be seen that the main objective of this is to minimize the damage due to the risk and take steps to prevent future recurrence of the risk.

**Proactive approaches**

The proactive approaches try to anticipate the possible risks that the project is susceptible to. After identifying the possible risks, actions are taken to eliminate the risks. If a risk cannot be avoided, these approaches suggest making plans to contain the effect of the risk.

For example, if man power turnover is anticipated (that is, some personnel may leave the project), then thorough documentation may be planned. Also, more than one developer may work on a work item and also some stand-by man power may be planned. Obviously, proactive approaches incur lower cost and time overruns when risk events occur and therefore is much more preferred by teams. However, when some risks cannot be anticipated, a reactive approach is usually followed.

## Risk Identification

Risk identification is somewhat similar to the project manager listing down his nightmares. For example, project manager 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.

- **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. It is very difficult to control something which cannot be seen. 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.
- **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 includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

### 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:

  The likelihood of a risk becoming real ($r$).

  The consequence of the problems associated with that risk ($s$).

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

- $p = r \times s$

  where, $p$ is the priority with which the risk must be handled, $r$ is the probability of the risk becoming real, and $s$ is the severity of damage caused due to the risk becoming real. 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 those risks.

## Risk Mitigation

After all the identified risks of a project have been assessed, plans are made to contain the most damaging and the most likely risks first. Different types of risks require different containment procedures. In fact, most risks require considerable ingenuity on the part of the project manager in tackling the risks.

There are three main strategies for risk containment:

- **Avoid the risk:** Risks can be avoided in several ways. Risks often arise due to project constraints and can be avoided by suitably modifying the constraints. The different categories of constraints that usually give rise to risks are:
- ***Process-related risk:*** These risks arise due to aggressive work schedule, budget, and resource utilisation.
- ***Product-related risks:*** These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability, etc.
- ***Technology-related risks:*** These risks arise due to commitment to use certain technology (e.g., satellite communication).

  A few examples of risk avoidance can be the following: Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.
- **Transfer the risk:** This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.
- **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 risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.
- For example, if you are using a compiler for recognizing user commands, you would have to construct a compiler for a small and very primitive command language first. There can be several strategies to cope up with a risk. To choose the most appropriate strategy for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. risk leverage = cost of reduction
- Even though we identified three broad ways to handle any risk, effective risk handling cannot be achieved by mechanically following a set procedure, but 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.

# UNIT-II

# CHAPTER-II

# Requirements Analysis and Specification

The requirements analysis and specification phase start after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.

The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the *software requirements specification* (SRS) document. The goal of the requirements analysis and specification phase can be stated in a nutshell as follows.

The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.

## REQUIREMENTS GATHERING AND ANALYSIS

The complete set of requirements are almost never available in the form of a single document from the customer. In fact, it would be unrealistic to expect the customers to produce a comprehensive document containing a precise description of what they want. Further, the complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be systematically gathered by the analyst from several sources in bits and pieces.

These gathered requirements need to be analyzed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources. We can conceptually divide the requirements gathering and analysis activity into two separate tasks:

- Requirements gathering
- Requirements analysis

### Requirements Gathering

Requirements gathering activity is also popularly known as *requirements elicitation*. The primary objective of the requirement's gathering task is to collect the requirements from the *stakeholders*.

Requirements gathering may sound like a simple task. However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents. Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.

Suppose a customer wants to automate some activity in his organisation that is currently being carried out manually. In this case, a working model of the system (that is, a manual system) exists. Availability of a working model is usually of great help in requirements gathering.

For example, if the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input and output forms and the details of the operational procedures. In this context, consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain office. In this case, the analyst would have to study the input and output forms and then understand how the outputs are produced from the input data. However, if a project involves developing something new for which no working model exists, then the requirements gathering activity becomes all the more difficult. In the absence of a working system, much more imagination and creativity is required on the part of the system analyst.

Typically, even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed. During visit to the customer site, the analysts normally interview the end-users and customer representatives,1 carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.

## Requirements Gathering Activities

**1. Studying the existing documentation:** The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the analyst. Typically, these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, and the broad category of features required.

**2. Interview:** Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each. For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants.

**3. Task analysis:** The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a *task*. We can therefore say that the software performs various tasks of the users. In this context, the analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to formulate the different steps necessary to realize the required functionality in consultation with the users.

**4. Scenario analysis:** A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different scenarios of a task, the behavior of the software can be different. For example, the possible scenarios for the book issue task of a library automation software may be:

- ▶ Book is issued successfully to the member and the book issue slip is printed.
- ▶ The book is reserved, and hence cannot be issued to the member.

- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.

**5. Form analysis:** Form analysis is an important and effective requirement gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis, the exiting forms and the formats of the notifications produced are analyzed to determine the data input to the system and the data that are output from the system. For the different sets of data input to the system, how the input data would be used by the system to produce the corresponding output data is determined from the users.

## Requirements Analysis

The main purpose of the requirements analysis activity is to analyze the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

For carrying out requirements analysis effectively, the analyst first needs to develop a clear grasp of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:

- What is the problem?
- Why is it important to solve the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the possible procedures that need to be followed to solve the problem?
- What are the likely complexities that might arise while solving the problem?

If there are external software or hardware with which the developed software has\ to interface, then what should be the data interchange formats with the external systems?

During requirements analysis, the analyst needs to identify\ and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

**Anomaly:** It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development. The following are two examples of anomalous requirements:

**Example:**

- While gathering the requirements for a process control application, the following requirement was expressed by a certain stakeholder: When the temperature becomes high, the heater should be switched off. Please note that words such as “high”, “low”, “good”, “bad”, etc. are indications of ambiguous requirements as these lack quantification and can be subjectively interpreted. If the threshold above which the temperature can be considered to be high is not specified, then it may get interpreted differently by different developers.
- **Inconsistency:** Two requirements are said to be inconsistent, if one of the requirements contradicts the other. The following are two examples of inconsistent requirements:

**Example:**

- Consider the following partial requirements that were collected from two different stakeholders in a process control application development project.
- The furnace should be switched-off when the temperature of the furnace rises above 500C
- When the temperature of the furnace rises above 500C, the water shower should be switched- on and the furnace should remain on.

The requirements expressed by the two stakeholders are clearly inconsistent.

- **Incompleteness:** An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much later, possibly while using the software. Often, incompleteness is caused by the inability of the customer to visualize the system that is to be developed and to anticipate all the features that would be required. An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

**Example:**

- In a chemical plant automation software, suppose one of the requirements is that if the internal temperature of the reactor exceeds 200°C then an alarm bell must be sounded. However, on an examination of all requirements, it was found that there is no provision for resetting the alarm bell after the temperature has been brought down in any of the requirements. This is clearly an incomplete requirement.

## Software Requirements Specification (SRS)

The Main aim of requirements specification:

  - Systematically organize the requirements arrived during requirements analysis.
  - Document requirements properly.

- The SRS document is useful in various contexts:
  - Statement of user needs

- Contract document
- Reference document
- Definition for implementation

- After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organize the requirements in the form of an SRS document. The SRS document usually contains all the user requirements in a structured though an informal form.
- Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write. One reason for this difficulty is that the SRS document is expected to cater to the needs of a wide variety of audience. In the following subsection, we discuss the different categories of users of an SRS document and their needs from it.

**Users of SRS Document**

Usually, a large number of different people need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

- **Users, customers, and marketing personnel:** These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs. Remember that the customer may not be the user of the software, but may be some one employed or designated by the user. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.
- **Software developers:** The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.
- **Test engineers:** The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.
- **User documentation writers:** The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.
- **Project managers:** The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.
- **Maintenance engineers:** The SRS document helps the maintenance engineers to understand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and code. Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.

Many software engineers in a project consider the SRS document to be a reference document. However, it is often more appropriate to think of the SRS document as the documentation of a contract between the development team and the customer. In fact, the SRS document can be used to resolve any disagreements between the developers and the customers that may arise in the future. The SRS document can even be used as a legal document to settle disputes between the customers and the developers in a court of law. Once the customer agrees to the SRS document, the development team proceeds to develop the software and ensure that it conforms to all the requirements mentioned in the SRS document.

## Characteristics of a Good SRS Document

The SRS document should describe the system (to be developed) as a black box, and should specify only the externally visible behaviour of the system. For this reason, the SRS document is also called the *black-box* specification of the software being developed.

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
- **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behavior of the system and not discuss the implementation issues. This view with which a requirement specification is written, has been shown in the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all.
- **Traceable:** It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*. Traceability is also important to verify the results of a phase with respect to the previous phase and to analyze the impact of changing a requirement on the design elements and the code.
- **Modifiable:** Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. Also, an SRS document is often modified after the project completes to accommodate future enhancements and evolution. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured. A well-structured document is easy to understand and modify.
- **Identification of response to undesired events:** The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

- **Verifiable:** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation. A requirement such as "the system should be user friendly" is not verifiable. On the other hand, the requirement—"When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out" is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

## Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems the most damaging problems are incompleteness, ambiguity, and contradictions.

There are many other types of problems that a specification document might suffer from. By knowing these problems, one can try to avoid them while writing an SRS document. Some of the important categories of problems that many SRS documents suffer from are as follows:

- **Over-specification:** It occurs when the analyst tries to address the "how to" aspects in the SRS document. For example, in the library automation problem, one should not specify whether the library membership records need to be stored indexed on the member's first name or on the library member's identification (ID) number. Over-specification restricts the freedom of the designers in arriving at a good design solution.
- **Forward references:** One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.
- **Wishful thinking:** This type of problems concern description of aspects which would be difficult to implement.
- **Noise:** The term noise refers to presence of material not directly relevant to the software development process. For example, in the register customer function, suppose the analyst writes that customer registration department is manned by clerks who report for work between 8 am and 5 pm, 7 days a week. This information can be called *noise* as it would hardly be of any use to the software developers and would unnecessarily clutter the SRS document, diverting the attention from the crucial points.
- Several other "sins" of SRS documents can be listed and used to guard against writing a bad SRS document and is also used as a checklist to review an SRS document.

## Important Categories of Customer Requirements

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

- Functional requirements
- Non-functional requirements
- Design and implementation constraints

- External interfaces required
- Other non-functional requirements
- Goals of implementation.

**Functional requirements**

The functional requirements capture the functionalities required by the users from the system. it is useful to consider a software as offering a set of functions $\{fi\}$ to the user. These functions can be considered similar to a mathematical function $f : I \rightarrow O$, meaning that a function transforms an element ($ii$) in the input domain ($I$) to a value ($oi$) in the output ($O$). This functional view of a system is shown schematically in Figure 4.1.

- Each function $f$i of the system can be considered as reading certain data $ii$, and then transforming a set of input data ($ii$) to the corresponding set of output data ($oi$). The functional requirements of the system should clearly describe each functionality that the system would support along with the corresponding input and output data set.

- Considering that the functional requirements are a crucial part of the SRS document, we discuss functional requirements how the functional requirements can be identified from a problem description. how the functional requirements can be documented effectively.

**Non-functional requirements**

- The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections. The remaining non-functional requirements should be documented later in a section and these should include the performance and security requirements.
- Design and implementation constraints: Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers.
- Some of the example constraints can be—corporate or regulatory policies that needs to be honored; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc.
- External interfaces required: Examples of external interfaces are—hardware, software and communication interfaces, user interfaces, report formats, etc. To specify the user interfaces, each interface between the software and the users must be described.
- The description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on.

- One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree. The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.
- Other non-functional requirements: This section contains a description of non-functional requirements that neither are design constraints and nor are external interface requirements.
- An important example is a performance requirement such as the number of transactions completed per unit time. Besides performance requirements, the other non-functional requirements to be described in this section may include reliability issues, accuracy of results, and security issues.

**<u>Functional Requirements</u>**

In order to document the functional requirements of a system, it is necessary to first learn to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements. The high-level functions would be split into smaller sub requirements.

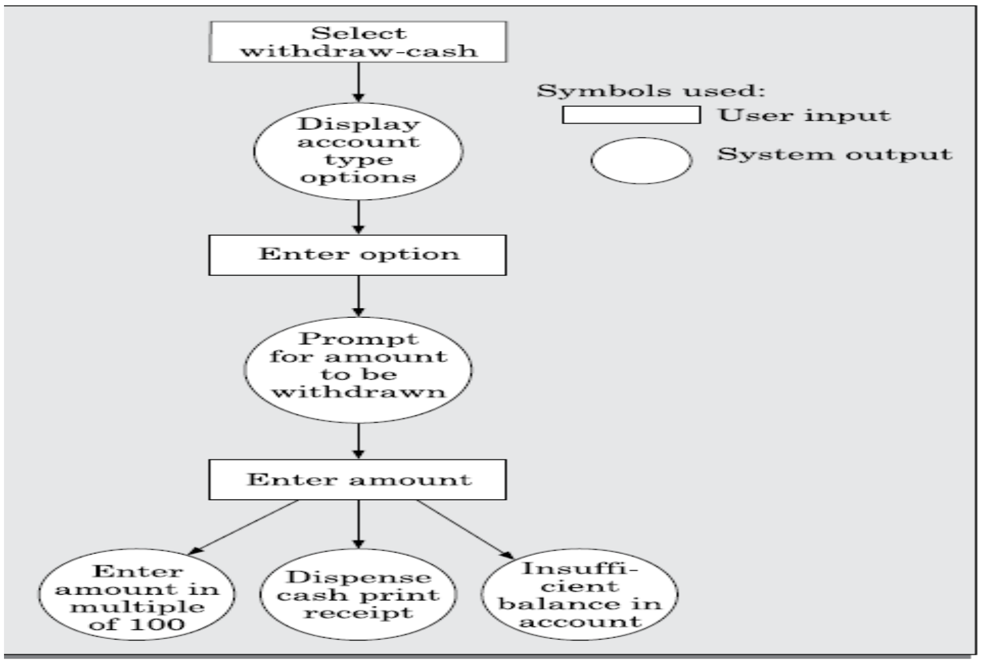
Each high-level function is an instance of use of the system (use case) by the user in some way. A high-level function is one using which the user can get some useful piece of work done.

For example, how useful must a piece of work be performed by the system for it to be called 'a useful piece of work'? Can the printing of the statements of the ATM transaction during withdrawal of money from an ATM be called a useful piece of work? Printing of ATM transaction should not be considered a high-level requirement, because the user does not specifically request for this activity. The receipt gets printed automatically as part of the withdraw money function. Usually, the user invokes (requests) the services of each high-level requirement.

- It may therefore be possible to treat print receipt as part of the withdraw money function rather than treating it as a high-level function. It is therefore required that for some of the high-level functions, we might have to debate whether we wish to consider it as a high-level function or not. However, it would become possible to identify most of the high-level functions without much difficulty after practicing the solution to a few exercise problems.
- Each high-level requirement typically involves accepting some data from the user through a user interface, transforming it to the required response, and then displaying the system response in proper format. For example, in a library automation software, a high-level functional requirement might be search-book.
- This function involves accepting a book name or a set of key words from the user, running a matching algorithm on the book list, and finally outputting the matched books. The generated system response can be in several forms, e.g., display on the terminal, a print out, some data transferred to the other systems, etc. However, in degenerate cases, a high-level requirement may not involve any data input to the system

or production of displayable results. For example, it may involve switch on a light, or starting a motor in an embedded application.

- Are high-level functions of a system similar to mathematical functions?
- For any given high-level function, there can be different interaction sequences or scenarios due to users selecting different options or entering different data items.
- The different scenarios occur depending on the amount entered for withdrawal. The different scenarios are essentially different behavior exhibited by the system for the same high-level function. Typically, each user input and the corresponding system action may be considered as a sub-requirement of a high-level requirement. Thus, each high-level requirement can consist of several sub-requirements.


Select withdraw-cash
Display account type options
Enter option
Prompt for amount to be withdrawn
Enter amount
Enter amount in multiple of 100
Dispense cash print receipt
Insuffi-cient balance in account
Symbols used:
User input
System output

2 User and system interactions in high-level functional requirement.

**Is it possible to determine all input and output data precisely?**

- In a requirements specification document, it is desirable to define the precise data input to the system and the precise data output by the system. Sometimes, the exact data items may be very difficult to identify. This is especially the case, when no working model of the system to be developed exists. In such cases, the data in a high-level requirement should be described using high-level terms and it may be very difficult to identify the exact components of this data accurately.
- Another aspect that must be kept in mind is that the data might be input to the system in stages at different points in execution. For example, consider the withdraw-cash function of an *automated teller machine* (ATM) of Figure 4.2. Since during the course of execution of the withdraw-cash function, the user would have to input the type of account, the amount to be withdrawn, it is very difficult to form a single high-level

name that would accurately describe both the input data. However, the input data for the subfunctions can be more accurately described

**<u>How to Identify the Functional Requirements?</u>**

- The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem.
- Remember that there can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to first identify the different types of users who might use the system and then try to identify the different services expected from the software by different types of users.
- The decision regarding which functionality of the system can be taken to be a high-level functional requirement and the one that can be considered as part of another function (that is, a subfunction) leaves scope for some subjectivity. For example, consider the issue-book function in a Library Automation System. Suppose, when a user invokes the issue-book function, the system would require the user to enter the details of each book to be issued. Should the entry of the book details be considered as a high-level function, or as only a part of the issue-book function? Many times, the choice is obvious. But sometimes it requires making non-trivial decisions.

**<u>How to Document the Functional Requirements</u>**

- Once all the high-level functional requirements have been identified and the requirements problems have been eliminated, these are documented. A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data.
- We now illustrate the specification of the functional requirements through two examples. Let us first try to document the withdraw-cash function of an *automated teller machine* (ATM) system in the following. The *withdraw-cash* is a high level requirement. It has several sub-requirements corresponding to the different user interactions.
- These user interaction sequences may vary from one invocation from another depending on some conditions. These different interaction sequences capture the different *scenarios*. To accurately describe a functional requirement, we must document all the different scenarios that may occur.

**Example: Withdraw cash from ATM**

***R.1: Withdraw cash***

*Description:* The withdraw-cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

- ***R.1.1: Select withdraw amount option***

*Input:* “Withdraw amount” option selected

*Output:* User prompted to enter the account type

- ***R.1.2: Select account type***

*Input:* User selects option from any one of the following—savings/checking/deposit.

*Output:* Prompt to enter amount

- ***R.1.3: Get required amount***

  *Input:* Amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

  *Output:* The requested cash and printed transaction statement.

  *Processing:* The amount is debited from the user’s account if sufficient balance is available, otherwise, an error message displayed.

- In order to properly identify the high-level requirements, a lot of common sense and the ability to visualize various scenarios that might arise in the operation of a function are required. Please note that when any of the aspects of a requirement, such as the state, processing description, next function to be executed, etc. are obvious, we have omitted it.
- We have to make a trade-off between cluttering the document with trivial details versus missing out some important descriptions.
- **Specification of large software:** If there are large number of functional requirements (much larger than seen), should they just be written in a long-numbered list of requirements?
- A better way to organize the functional requirements in this case would be to split the requirements into sections of related requirements. For example, the functional requirements of an academic institute automation software can be split into sections such as accounts, academics, inventory, publications, etc. When there are too many functional requirements, these should be properly arranged into sections.
- For example, the following can be sections in the trade house automation software:
- Customer management
- Account management
- Purchase management
- Vendor management
- Inventory management
- **Level of details in specification:** Even for experienced analysts, a common dilemma is in specifying too little or specifying too much. In practice, we would have to specify only the important input/output interactions in a functionality along with the processing required to generate the output from the input.

- However, if the interaction sequence is specified in too much detail, then it becomes an unnecessary constraint on the developers and restricts their choice in solution. On the other hand, if the interaction sequence is not sufficiently detailed, it may lead to ambiguities and result in improper implementation.
- **Traceability**

  Traceability means that it would be possible to identify (trace) the specific design component which implements a given requirement, the code part that corresponds to a given design component, and test cases that test a given requirement. Thus, any given code component can be traced to the corresponding design component, and a design component can be traced to a specific requirement that it implements and *vice versa*.

  Traceability analysis is an important concept and is frequently used during software development. For example, by doing a traceability analysis, we can tell whether all the requirements have been satisfactorily addressed in all phases. It can also be used to assess the impact of a requirements change. That is, traceability makes it easy to identify which parts of the design and code would be affected, when certain requirement change occurs. It can also be used to study the impact of a bug that is known to exist in a code part on various requirements, etc.
- To achieve traceability, it is necessary that each functional requirement should be numbered uniquely and consistently. Proper numbering of the requirements makes it possible for different documents to uniquely refer to specific requirements.
- An example scheme of numbering the functional requirements is shown in Examples 4.7 and 4.8, where the functional requirements have been numbered R.1, R.2, etc. and the sub requirements for the requirement R.1 have been numbered R.1.1, R.1.2, etc.

## Organization of the SRS Document

- The organization of an SRS document as prescribed by the IEEE 830 standard [IEEE 830]. Please note that IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it, as may be required for specific projects.
- Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged as may be considered prudent by the analyst. However, organization of the SRS document to a large extent depends on the preferences of the system analyst himself, and he is often guided in this by the policies and standards being followed by the development company.
- Also, the organization of the document and the issues discussed in it to a large extent depend on the type of the product being developed. However, irrespective of the company's principles and product type, the three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.
- The introduction section should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental

characteristics. The introduction section may include the hardware that the system will run on, the devices that the system will interact with and the user skill-levels.

- Description of the user skill-level is important, since the command language design and the presentation styles of the various documents depend to a large extent on the types of the users it is targeted for. For example, if the skill-levels of the users is "novice", it would mean that the user interface has to be very simple and rugged, whereas if the user-level is "advanced", several short cut techniques and advanced features may be provided in the user interface.
- It is desirable to describe the formats for the input commands, input data, output reports, and if necessary, the modes of interaction. We have already discussed how the contents of the Sections on the functional requirements, the non-functional requirements, and the goals of implementation should be written. In the following subsections, we outline the important sections that an SRS document should contain as suggested by the IEEE 830 standard, for each section of the document, we also briefly discuss the aspects that should be discussed in it.
- **Introduction**

  **Purpose:** This section should describe where the software would be deployed and how the software would be used.

  **Project scope:** This section should briefly describe the overall context within which the software is being developed. For example, the parts of a problem that are being automated and the parts that would need to be automated during future evolution of the software.

  **Environmental characteristics:** This section should briefly outline the environment (hardware and other software) with which the software will interact.
- **Overall description of organization of SRS document**

  **Product perspective:** This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing system, or it is a new software. If the software being developed would be used as a component of a larger system, a simple schematic diagram can be given to show the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

  **Product features:** This section should summarize the major ways in which the software would be used. Details should be provided in Section 3 of the document. So, only a brief summary should be presented here.
- **User classes:** Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities that they are expected to invoke, or their levels of expertise in using computers.
- **Operating environment:** This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.

- **Design and implementation constraints:** In this section, the different constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.
- **User documentation:** This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.
- **Functional requirements**

  This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.

  1. User class 1

  (a) Functional requirement 1.1

  (b) Functional requirement 1.2

  2. User class 2

  (a) Functional requirement 2.1

  (b) Functional requirement 2.2
- **External interface requirements**
- **User interfaces:** This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard push buttons (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, etc. The details of the user interface design should be documented in a separate user interface specification document.
- **Hardware interfaces:** This section should describe the interface between the software and the hardware components of the system. This section may include the description of the supported device types, the nature of the data and control interactions between the software and the hardware, and the communication protocols to be used.
- **Software interfaces:** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries, and integrated commercial components, etc. Identify the data items that would be input to the software and the data that would be output should be identified and the purpose of each should be described.
- **Communications interfaces:** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc. This section should define any pertinent

message formatting to be used. It should also identify any communication standards that will be used, such as TCP sockets, FTP, HTTP, or SHTTP. Specify any communication security or encryption issues that may be relevant, and also the data transfer rates, and synchronisation mechanisms.

- **Other non-functional requirements for organization of SRS document**

This section should describe the non-functional requirements other than the design and implementation constraints and the external interface requirements that have been described

- **Performance requirements:** Aspects such as number of transactions to be completed per second should be specified here. Some performance requirements may be specific to individual functional requirements or features. These should also be specified here.
- **Safety requirements:** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here. For example, recovery after power failure, handling software and hardware failures, etc. may be documented here.
- **Security requirements:** This section should specify any requirements regarding security or privacy requirements on data used or created by the software. Any user identity authentication requirements should be described here. It should also refer to any external policies or regulations concerning the security issues. Define any security or privacy certifications that must be satisfied.
- For software that have distinct modes of operation, in the functional requirements section, the different modes of operation can be listed and, in each mode, the specific functionalities that are available for invocation can be organized as follows.
- **Functional requirements**

  1. Operation mode 1

- (a) Functional requirement 1.1
- (b) Functional requirement 1.2

  2. Operation mode 2

- (a) Functional requirement 2.1

  (b) Functional requirement 2.2