UNIT-II

2.1 Software Project Management

Effective project management is crucial to the success of any software development project. In the past, several projects have failed not for want of competent technical professionals neither for lack of resources, but due to the use of faulty project management practices. Therefore, it is important to carefully learn the latest software project management techniques.

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

Project management involves use of a set of techniques and skills to steer a project to success. Before focusing on these projects management techniques, let us first figure out who should be responsible for managing a project. A project manager is usually an experienced member of the team who essentially works as the administrative leader of the team. For small software development projects, a single member of the team assumes the responsibilities for both project management and technical management. For large projects, a different member of the team (other than the project manager) assumes the responsibility of technical leadership. The responsibilities of the technical leader includes addressing issues such as which tools and techniques to use in the project, high-level solution to the problem, specific algorithms to use, etc.

2.1.1 Software Project Management Complexities

Management of software projects is much more complex than management of many other types of projects. The main factors contributing to the complexity of managing a software project, as identified by [Brooks75], are the following:

Invisibility: Software remains invisible, until its development is complete and it is operational. Anything that is invisible is difficult to manage and control. Consider a house building project. For this project, the project manager can very easily assess the progress of the project through a visual examination of the building under construction. Therefore, the manager can closely monitor the progress of the project, and take remedial actions whenever he finds that the progress is not as per plan. In contrast, it becomes very difficult for the manager of a software project to assess the progress of the project due to the invisibility of software. The best that he can do perhaps is to monitor the milestones that have been completed by the development team and the documents that have been produced—which are rough indicators of the progress achieved.

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: Because the software part of any system is easier to change as compared to the hardware part, the software part is the one that gets most frequently changed. This is especially true in the later stages of a project. As far as hardware development is concerned, any late changes to the specification of the hardware system under development usually amounts to redoing the entire project. This makes late changes to a hardware project prohibitively expensive to carry out. This possibly is a reason why requirement changes are frequent in software projects. These changes usually arise from changes to the business practices, changes to the hardware or underlying software (e.g. operating system, other applications), or just because the client changes his mind.

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 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. Due to the inherent complexity of the functioning of a software product in terms of the basic parts making up the software, many types of risks are associated with its development. This makes managing these projects much more difficult as compared to many other kinds of projects.

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 others he had encountered in the past. As a result, a software project manager has to confront many unanticipated issues in almost every project that he manages.

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. This requirement not only makes it difficult to get a software product up and working, but also makes reusing parts of one software product in another difficult. This requirement of exact conformity of the parameters of a function introduces additional risks and contributes to the complexity of managing software projects.

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 labour-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 multi-storeyed building, etc. In a software development project, the life cycle activities not only highly intellect-intensive, but each member has to typically interact, review, and interface with several other members, constituting another dimension of complexity of software projects.

2.1.2 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 responsibility of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations. Most managers take the responsibilities for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation, and interfacing with clients. These activities are certainly numerous and varied. We can still broadly classify these activities into two major types—project planning and project monitoring and control.

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.

Project planning involves estimating several characteristics of a project and then planning the project activities based on these estimates made.

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

Project planning is undertaken and completed before any development activity starts.

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 organisation 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 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.

Size is the most fundamental parameter based on which all other estimations and project plans are made.

As can be seen from Figure, 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.

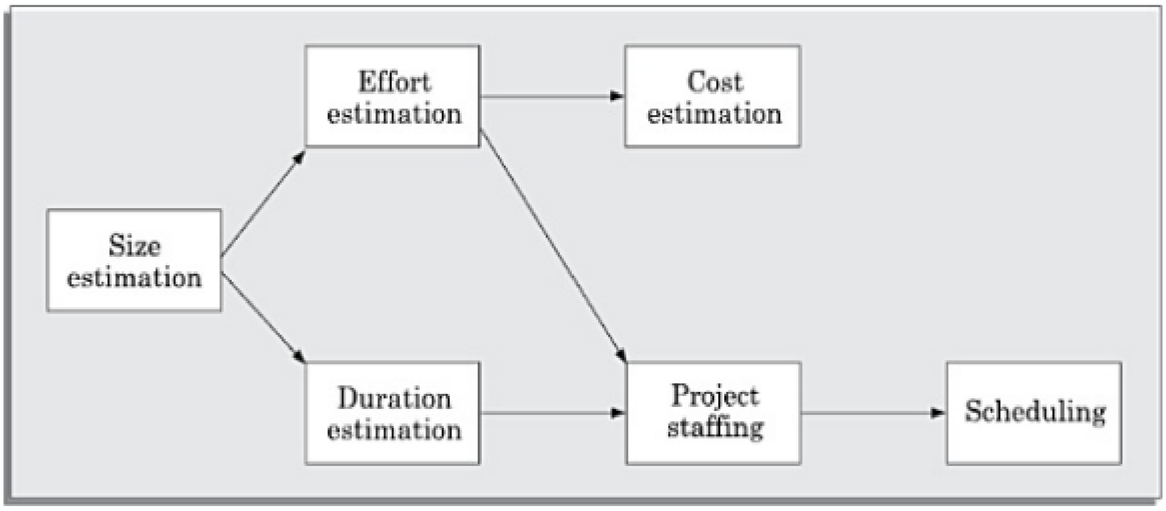

Figure: Precedence ordering among planning activities.

Project monitoring and control: Project monitoring and control activities are undertaken once the development activities start.

The focus of project monitoring and control activities is to ensure that the software development proceeds as per plan.

While carrying out project monitoring and control activities, a project manager may sometimes find it necessary 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.

2.1.3 Metrics for Project Size Estimation

As already mentioned, accurate estimation of project size is central to satisfactory estimation of all other project parameters such as effort, completion time, and total project cost. Before discussing the available metrics to estimate the size of a project, let us examine what the term "project size" exactly means. The size of a project is obviously not the number of bytes that the source code occupies, neither is it the size of the executable code.

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

Currently, two metrics are popularly being used to measure size—lines of code (LOC) and function point (FP). Each of these metrics has its own advantages and disadvantages. These are discussed in the following subsection. 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 so on, until the LOC of the leaf-level modules are small enough to be predicted. To be able to predict the LOC count for the various leaf-level modules sufficiently accurately, past experience in developing similar modules is very helpful. By adding the estimates for all leaf level modules together, project managers arrive at the total size estimation. In spite of its conceptual simplicity, LOC metric has several shortcomings when used to measure problem size.

From the project manager's perspective, the biggest shortcoming of the LOC metric is that the LOC count is very difficult to estimate during project planning stage, and can only be accurately computed after the software development is complete.

Function Point (FP) Metric

Function point metric was proposed by Albrecht in 1983. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, 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 product has fully been developed.

Conceptually, the function point metric is based on the idea that a software product supporting many features would certainly be of larger size than a product with less number of features.

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. 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.

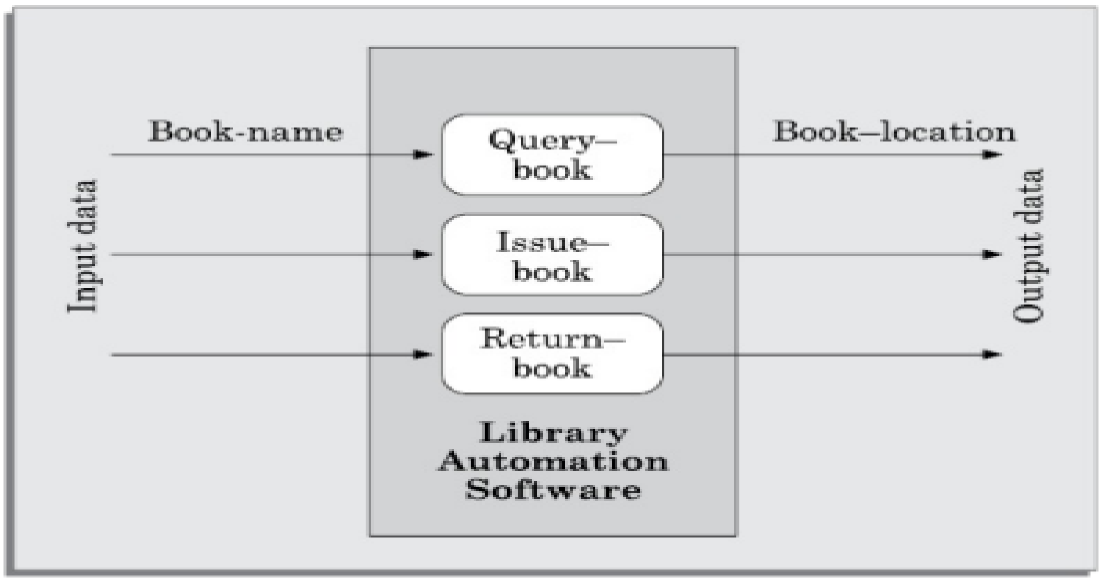

Figure: System function as a mapping of input data to output data

For example, the query book feature (see Figure) 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.

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.

2.1.4 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

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 formalised to a large extent.

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 multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter.

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.

2.1.5 Empirical Estimation Techniques

These techniques are easy to use and give reasonably accurate estimates. Two popular empirical estimation techniques are:

- Expert judgement and
- Delphi estimation techniques

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 analysing the problem thoroughly.

Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. However, this technique suffers from several shortcomings. The outcome of the expert judgement technique is subject to human errors and individual bias.

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 minimised 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.

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 co-ordinator. In this approach, the co-ordinator 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 co-ordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The co-ordinator 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 are 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 co-ordinator 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.

2.1.6 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.

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.

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 programs 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.

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. 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 provides different sets of 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 considered to be an appropriate unit for measuring effort, because developers are typically assigned to a project for a certain number of months.

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.

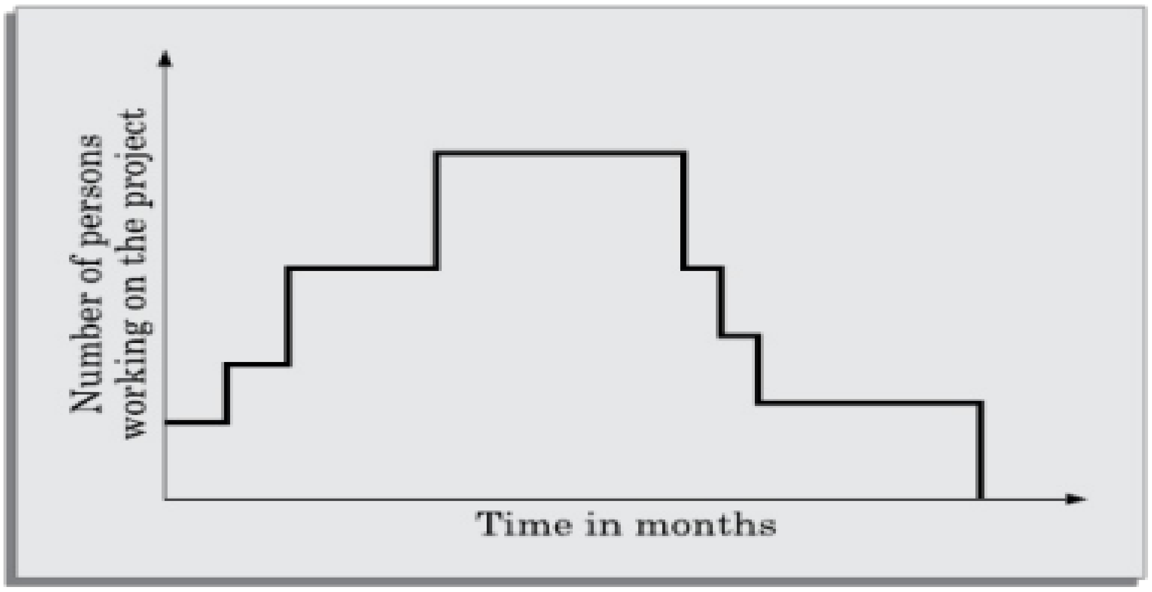

Figure: Person-month curve

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 × (KLOC)a_2 PM

Tdev = b1 × (Effort)b_2 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).

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.

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.

Example Problem

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of a software developer is Rs. 15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.

From the basic COCOMO estimation formula for organic software: Effort = 2.4 × (32)1.05 = 91 PM

Nominal development time = 2.5 × (91)0.38 = 14 months

Staff cost required to develop the product = 91 × Rs. 15, 000 = Rs. 1,465,000

Intermediate COCOMO

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort as well as the time required to develop the product. For example the effort to develop a product would vary depending upon the sophistication of the development environment.

Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognises this fact and refines the initial estimates.

The intermediate COCOMO model uses a set of 15 cost drivers (multipliers) that are determined based on various attributes of software development. These cost drivers are multiplied with the initial cost and effort estimates (obtained from the basic COCOMO) to appropriately scale those up or down. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, the initial estimates are scaled upward. Boehm requires the project manager to rate 15 different parameters for a particular project on a scale of one to three. For each such grading of a project parameter, he has suggested appropriate cost drivers (or multipliers) to refine the initial estimates.

In general, the cost drivers identified by Boehm can be classified as being attributes of the following items:

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

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required, etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, their programming capability, analysis capability, etc.

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.

Complete COCOMO

A major shortcoming of both the basic and the intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up of several smaller sub-systems. These sub-systems often have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some even embedded. Not only may the inherent development complexity of the subsystems be different, but for some subsystem the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on.

The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual sub-systems.

Let us consider the following development project as an example application of the complete COCOMO model. A distributed management information system (MIS) product for an organisation having offices at several places across the country can have the following sub-component:

• Database part

• Graphical user interface (GUI) part

• Communication part

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

To further improve the accuracy of the results, the different parameter values of the model can be fine-tuned and validated against an organisation’s historical project database to obtain more accurate estimations. Estimation models such as COCOMO are not totally accurate and lack a full scientific justification. Still, software cost estimation models such as COCOMO are required for an engineering approach to software project management.

2.1.7 Halstead's Software Science—An Analytical Technique

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

For a given program, let:

- h1 be the number of unique operators used in the program,
- h2 be the number of unique operands used in the program,
- N1 be the total number of operators used in the program,
- N2 be the total number of operands used in the program.

There is no general agreement among researchers on what is the most meaningful way to define the operators and operands for different programming languages. However, a few general guidelines regarding identification of operators and operands for any programming language can be provided. For instance, assignment, arithmetic, and logical operators are usually counted as operators. A pair of parentheses, as well as a block begins —blocks end pair, are considered as single operators. A label is considered to be an operator, if it is used as the target of a GOTO statement.

Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program.

Thus, length N = N1 + N2.

Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notion of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program.

Thus, program vocabulary h = h1 + h2.

Program Volume

The length of a program (i.e., the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used.

Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 h$$

Let us try to understand the important idea behind this expression. Intuitively, the program volume V is the minimum number of bits needed to encode the program. In fact, to represent h different identifiers uniquely, we need at least log2 h bits (where h is the program vocabulary). In this scheme, we need N log2 h bits to store a program of length N. Therefore, the volume V represents the size of the program by approximately compensating for the effect of the programming language used.

Potential Minimum Volume

The potential minimum volume V* is defined as the volume of the most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction, say a function call like foo ();. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands. Note that the operands are the input and output data items.

Thus, if an algorithm operates on input and output data d1, d2, ... dn, the most succinct program would be f(d1, d2 , ..., dn); for which, h1 = 2, h2 = n.

Therefore, $V^* = (2 + h2) \log_2 (2 + h2)$.

Effort and Time

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

Thus, the programming effort $E = V^2/V^*$ (since $L = V^*/V$) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

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

Length Estimation

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

Halstead assumed that it is quite unlikely that a program has several identical parts— in formal language terminology identical substrings—of length greater than h(h being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is usually made into a procedure or a function. Thus, we can safely assume that any program of length N consists of N/h unique strings of length h. Now, it is a standard combinatorial result that for any given alphabet of size K, there are exactly K r different strings of length r. Thus,

$$\frac{N}{\eta} \leq \eta^{\eta}$$

or

$$N \leq \eta^{\eta+1}$$

2.1.8 Risk Management

A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway. 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 Identification

The project manager needs to anticipate the risks in a project as early as possible. As soon as a risk is identified, effective risk management plans are made, so that the possible impacts of the risks is minimised. So, early risk identification is important. 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 organisation, etc. All such risks that are likely to affect a project must be identified and listed.

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

- Project Risks
- Technical Risks, and
- Business Risks

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.

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:

$$p = r * s$$

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 prioritised, 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).

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 are to build a prototype that tries out the technology that you are trying to use.

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. For this we may compute the risk leverage of the different risks. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

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

Requirements Analysis and Specification

The requirements analysis and specification phase starts 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.

Who carries out requirements analysis and specification?

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather a n d analyse customer requirements and then write the requirements specification document are known as system analysts in the software industry parlance.

System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the software requirements specification (SRS) document.

The SRS document is the final outcome of the requirements analysis and specification phase.

How is the SRS document validated?

Once the SRS document is ready, it is first reviewed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete. The SRS document is then given to the customer for review. After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

What are the main activities carried out during requirements analysis and specification phase?

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

• Requirements gathering and analysis

• Requirements specification

2.2.1 Requirements Gathering and Analysis

The complete sets 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 he wants. Further, the complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be gathered by the analyst from several sources in bits and pieces. These gathered requirements need to be analysed 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

2.2.1.1 Requirements Gathering

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

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, the 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 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.

Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete. In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

1. Studying 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 developers. Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.

2. Interview: Typically, there are many different categories of users of 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 library automation software could be the library members, the librarians, and the accountants. The library members would like to use the software to query availability of books and issue and return books. The librarians might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc. The accounts personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.

3. Task analysis: The users usually have a black-box view of software and consider the software as something that provides a set of services (functionalities). A service supported by 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 realise the required functionality in consultation with the users.

2.2.1.2 Requirements Analysis

After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements. It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness, since each stakeholder typically has only a partial and incomplete view of the software. Therefore, it is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.

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?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.

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.

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other.

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 visualise 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 requirement.

2.2.2 Software Requirements Specification (SRS)

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 organise 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.

2.2.2.1 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.

2.2.2.2 Why Spend Time and Resource to Develop an SRS Document?

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work.

Forms an agreement between the customers and the developers: A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

Provides a basis for estimating costs and schedules: Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The project manager also uses the SRS document for work scheduling.

Provides a baseline for validation and verification: The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the test plan.

Facilitates future extensions: The SRS document usually serves as a basis for planning future enhancements.

2.2.2.3 Characteristics of a Good SRS Document

The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. IEEE Recommended Practice for Software Requirements Specifications [IEEE830] describes the content and qualities of a good software requirements specification (SRS). Some of the identified desirable qualities of an SRS document are the following:

Concise: The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase 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 behaviour of the system and not discuss the implementation issues.

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 analyse 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.

2.2.2.4 Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems. As discussed earlier, the most damaging problems are incompleteness, ambiguity, and contradictions. There are many other types 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 t h e register customer function, suppose the analyst writes that customer registration department is manned by clerks who report for work between 8am and 5pm, 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.

2.2.2.5 Important Categories of Customer Requirements

A good SRS document should properly categorize and organise the requirements into different sections [IEEE830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following.

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

- Functional requirements
- Non-functional requirements
    - Design and implementation constraints
    - External interfaces required
    - Other non-functional requirements
- Goals of implementation.

2.2.3 Formal System Specification

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realisable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by its specification language. More precisely, a formal specification language consists of two sets: syn and sem, and a relation sat between them.

The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn, and model of the system sem, if sat (syn, sem), then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

The different stages in this system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.

Model versus property-oriented methods

Formal methods are usually classified into two broad categories: the so called model-oriented and the property-oriented approaches. In a model oriented style, one defines a system's behaviour directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the property-oriented style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Let us consider a simple producer/consumer example. In a property-oriented style, we would probably start by listing the properties of the system like: the consumer can start consuming only after the producer has produced an item; the producer starts to produce an item only after the consumer has consumed the last item, etc.

Two examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In a model-oriented style, we would start by defining the basic operations, p (produce) and c (consume). Then we can state that $S\ 1 + p \Rightarrow S$, $S + c \Rightarrow S\ 1$.

Thus model-oriented approaches essentially specify a program by writing another, presumably simpler program.

2.2.3.1 Operational Semantics

Informally, the operational semantics of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a single run of the system and how the runs are grouped together to describe the behaviour of the system.

2.2.4 Axiomatic Specification

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

How to develop an axiomatic specification?

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

- Establish the range of input values over which the function should behave correctly. Establish the constraints on the input parameters as a predicate.
- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.

- Establish the changes made to the function’s input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type assertion is not necessary for pure functions.
- Combine all of the above into pre- and post-conditions of the function.

2.2.5 Algebraic Specification

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980, 1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Essentially, algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations defined in this set; e.g. {I, +, -, *, / }. In contrast, alphabetic strings S together with operations of concatenation and length {S, I, con, len}, is not a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in a heterogeneous algebra is called a sort of the algebra. To define a heterogeneous algebra, besides defining the sorts, we need to specify the involved operations, their signatures, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using equations. An algebraic specification is usually presented in four sections.

Types section: In this section, the sorts (or the data types) being used is specified.

Exception section: This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

Syntax section: This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the signature of the operator. For example, PUSH takes a stack and an element as its input and returns a new stack that has been created.

Equations section: This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

2.2.6 Executable Specification and 4GL

When the specification of a system is expressed formally or is described by using a programming language, then it becomes possible to directly execute the specification without having to design and write code for implementation. However, executable specifications are usually slow and inefficient, 4GLs (4th Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of large granularity commonality across data processing applications which have been identified and mapped to program code. 4GLs get their power from software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in 3GLs results in up to 50 per cent lower memory usage and also the program execution time can reduce up to ten folds.