This prototype is further enhanced by the developer with better understanding of the requirements and preparation of a final specification document. This working prototype is evaluated by the customer and the feedback received helps the developers to get rid of the uncertainties in the requirements and to start a re-iteration of requirements for further clarification. The prototype can be a usable program with limited functionality but cannot be used as a final product. This prototype is thrown away after preparing the final SRS; however the understanding obtained from developing the prototype helps in developing the actual system.

The development of prototype is an additional cost overhead but still the total cost is lower than that of the software developed using a waterfall model. The earlier the prototype is developed the speedier would be the software development process. This model involves a lot of customer interaction which is not always possible.

2.2.4 Iterative Enhancement Model

This model is similar to the waterfall model in the sense that individual phases of software development will be carried using the waterfall model but the entire product is available after many cycles. At the end of each cycle a product is released with additional functionality in later releases.

The SRS prepared by the customer and the developer after the requirement analysis phase contains as many requirements as possible. These requirements are prioritized by the developer and implemented in multiple cycles of design, build and testing as mentioned in Figure 2.4.

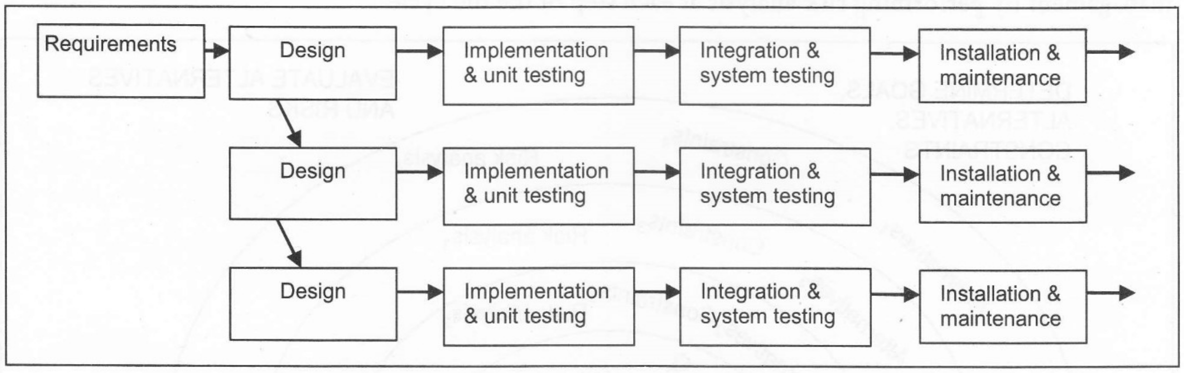
Requirements
Design
Implementation & unit testing
Integration & system testing
Installation & maintenance
Design
Implementation & unit testing
Integration & system testing
Installation & maintenance
Design
Implementation & unit testing
Integration & system testing
Installation & maintenance

Figure 2.4: Iterative Enhancement Model

2.2.5 Evolutionary Development Model

This model is similar to the iterative enhancement model. It involves the same phases as those in waterfall model but in a cyclical fashion. It differs from the iterative enhancement model as it does not require a usable product at the end of each cycle. Here requirements are implemented through category and not priority. It is useful when it is not required to deliver a minimal version of the system quickly, when the technology is relatively new, project is complex and the entire functionality must be delivered at once although the requirements are unstable and unclear at the beginning.

2.2.6 Spiral Model

The classical models do not deal with the uncertainty with the software projects. A lot risk assessment and analysis form a part of the software development. This was first realized by Barry Boehm, who introduced the factor of "project risk" into the life cycle model which resulted in the spiral model in 1986 as shown in Figure 2.5.

The radial dimension represents the cumulative cost and each path around the spiral indicates the incremented cost. The angular dimensions depict the progress made in each cycle completion. Each loop of the spiral, clockwise from the X-axis, through 360 degrees represents one phase. Each phase is split into four sectors namely:

- *Planning:* Determining aims, alternatives and constraints.
- *Risk Analysis:* Analyzing alternatives and identify and resolution of risks.
- *Development:* Product development and testing.
- *Assessment:* Customer review.

The essential concept of the Spiral Model is "to minimize risks by the repeated use of prototypes [emphasis added] and other means. Unlike other models, at every stage risk analysis is performed. The Spiral Model works by building progressively more complete versions of the software by starting at the center of the spiral and working outwards. With each loop of the spiral, the customer evaluates the work and suggestions are made for its modification. Additionally, with each loop of the spiral, a risk analysis is performed which results in a 'go/no-go' decision. If the risks are determined to be too great then the project is terminated" [Frankovich 1998]. Thus, the Spiral Model addresses the problem of requirements engineering through development of prototypes, and it addresses the need for risk management by performing risk analysis at each step of the life cycle.

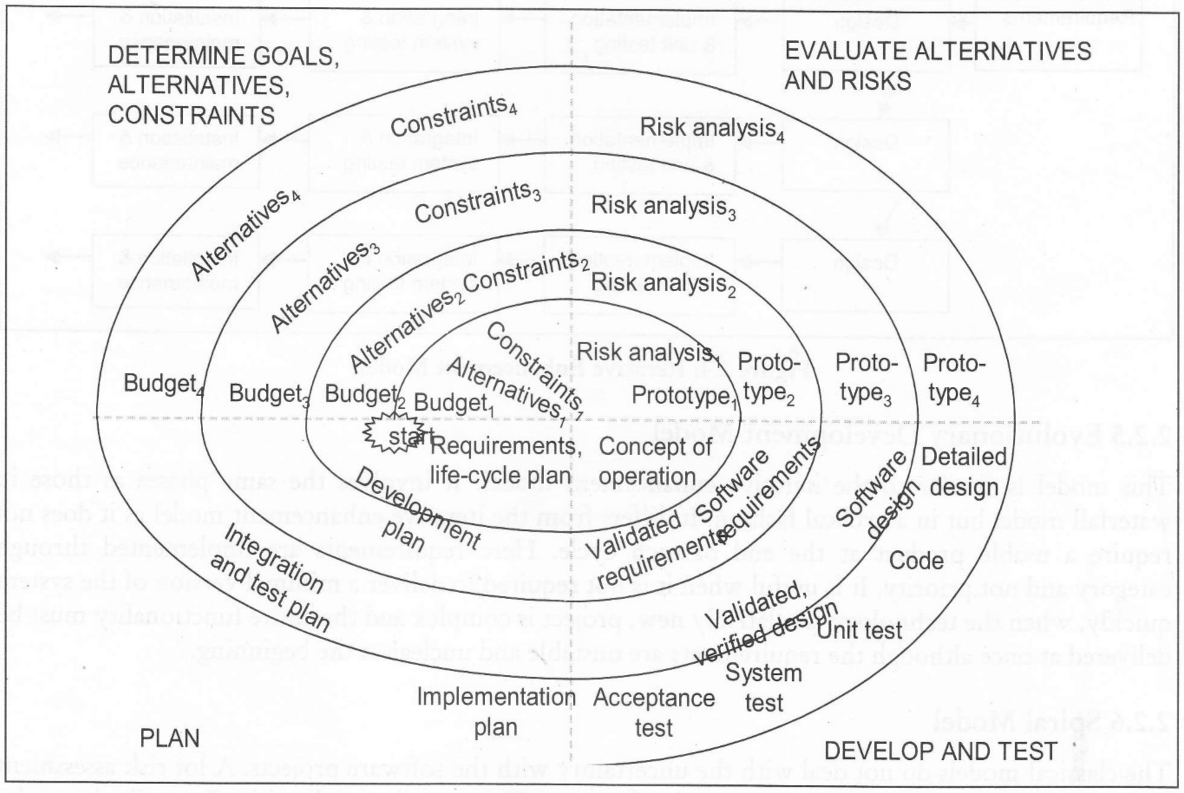

Figure 2.5: Spiral Model

2.3 CAPABILITY MATURITY MODEL

The Capability Maturity Model (CMM) is a software process improvement strategy. It does not depend upon the actual life cycle model used for software development. It was developed by Software Engineering Institute (SEI) of Carnegie-Mellon University in 1986.CMM is used to judge the order to enhance these maturity levels maturity levels of the processes of an organization and identify the key practices that must be followed in the various maturity levels are as shown in Figure 2.6.

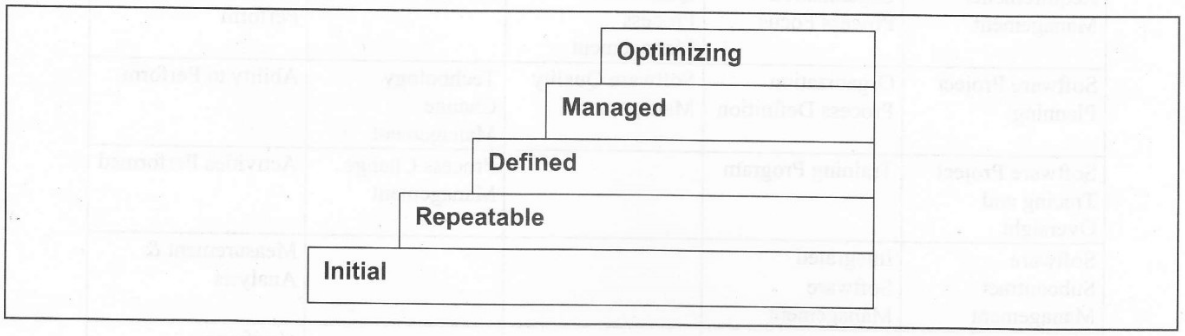

Figure 2.6: Maturity Levels of CMM

- *Initial (Maturity Level 1):* At the initial level, processes are disorganized, even chaotic and are done on an ad-hoc basis. Success is likely to depend on individual efforts, and is not considered to be repeatable, because processes would not be sufficiently defined and documented to allow them to be replicated. Software processes are unpredictable and activities carried out are a result of crisis and are not pre-planned.
- *Repeatable (Maturity Level 2):* At the repeatable level, basic project management techniques are established, and successes could be repeated, because the requisite processes would have been made established, defined, and documented. The management practices are based on the results of the past projects and current requests. It employs sound engineering and management practices.
- *Defined (Maturity Level 3):* At the defined level, an organization has developed its own standard software process through greater attention to documentation, standardization, and integration. Future projects are designed as per these standards and hence, the cost, schedule and functionality are under control.
- *Managed (Maturity Level 4):* At the managed level, an organization monitors and controls its own processes through data collection and analysis. The organizations set quantitative goals for the software processes and the processes are instrumented accordingly. As such the processes are predictable and the resultant product is of high quality.
- *Optimizing (Maturity Level 5):* At the optimizing level, processes are constantly being improved through monitoring feedback from current processes and introducing innovative processes to better serve the organization's particular needs. The focus lies on identifying strengths and weaknesses to prevent defects in the long run.

A complete maturity level takes 18 months to 3 years to advance, but moving from level 1 to level 2 sometimes takes 3 or even 5 years.

Key Process Areas

Each maturity level, except level 1, can be further divided into several key process areas (KPAs) which an organization must conform to for software process improvement. These are tabulated in Table 2.1 below: Key process areas are grouped in the stages/levels from 2 to 5.

Table 2.1: CMM KPAs

Level 2	Level 3	Level 4	Level 5	Common Features
Requirements Management	Organization Process Focus	Quantitative Process Management	Defect Prevention	Commitment to Perform
Software Project Planning	Organization Process Definition	Software Quality Management	Technology Change Management	Ability to Perform
Software Project Tracing and Oversight	Training Program		Process Change Management	Activities Performed
Software Subcontract Management	Integrated Software Management			Measurement & Analysis
Software Quality Assurance	Software Product Engineering			Verifying Implementation
Software Configuration Management	Inter group Coordination			
	Peer Reviews			

It has been stipulated by a lot of software organizations that wish to work as a subcontractor must conform to CMM level 3 or higher. This has lead to the growth in the importance of CMM.

2.4 ISO

"International Organization of Standardization" Instead of using an acronym (IOS) they used the Greek word for equal, which is ISO.

2.4.1 ISO 9000

The SEI CMM is an attempt to improve software quality by improving the underlying software processes. Another attempt based on International Standards Organization (ISO) 9000 series is based on software quality improvement. This standard although being used in over 130 countries is not industry specific and can be applied to a wide range of products e.g. automobiles, televisions, refrigerators, etc. Thus, we can conclude that ISO 9000 series is a set of documents dealing with quality systems that can be used for software quality assurance purposes. Within the ISO 9000 series, standard ISO 9001 is most applicable to software development.

Contrasting ISO 9001 and CMM

Although ISO 9001 and CMM are related in a lot of ways there are some issues that are covered in one but not in the other. These differences are listed in Table 2.2.

Table 2.2: Differences between ISO 9001 and CMM

ISO 9001	CMM
It determines the minimum criteria for an acceptable quality system.	It emphasize on continuous process improvement.
ISO has a wider scope including software, hardware, materials and services.	ISO focuses strictly on software.

2.4.2 ISO 9001

ISO 9001 is a standard for quality management systems and CMMI is a model for process improvement. If an ISO-certified organization wishes to improve its processes continuously, implementing CMMI would be a good choice, as it provides more detailed practices for process improvement than the ISO standards. However, there are two issues that need to be resolved when an ISO-certified organization implements CMMI. First, it is not easy to identify any reusable parts of the ISO standards, and it would be advantageous to be able to reuse selected portions of the ISO standards during CMMI adoption in order to use existing resources to their best advantage. Second, it is difficult for an ISO-certified organization to implement CMMI in a straightforward, easy manner because of the differences in the language, structure, and details of the two sets of documents. In this paper, we present our unified model for ISO 9001:2000 and CMMI that resolves these two issues. Our model would be an extremely useful tool for ISO-certified organizations that plan to implement CMMI.

According to the Carnegie Mellon University Software Engineering Institute, CMM is a common-sense application of software or business process management and quality improvement concepts to software development and maintenance. It's a community-developed guide for evolving towards a culture of engineering excellence, model for organizational improvement. The underlying structure for reliable and consistent software process assessments and software capability evaluations.

The Capability Maturity Model for Software (CMM) is a framework that describes the key elements of an effective software process. There are CMM's for non software processes as well, such as Business Process Management (BPM). The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process. The CMM covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality. The CMM establishes a yardstick against which it is possible to judge, in a repeatable way, the maturity of an organization's software process and compare it to the state of the practice of the industry. The CMM can also be used by an organization to plan improvements to its software process. It also reflects the needs of individuals performing software process, improvement, software process assessments, or software capability evaluations; is documented; and is publicly available.

2.4.3 ISO 9002

ISO 9002 is no longer in use. It was the standard that applied to organizations that did not do design or development. It was made obsolete with the 2000 year revisions. Now companies that do not do design are registered to ISO 9001; they include a "Permissible Exclusion" in the Quality Manual stating that design and development do not apply and are not included in the Quality System. ISO 9000 is often used to refer to the set of ISO Quality Management System documents. ISO 9001 is the document that contains the requirements for the Quality Management System. You will register to ISO 9001. ISO

9000 is a guidance document on the fundamentals and vocabulary for quality management systems. ISO 9002 and ISO 9003 are no longer in use. All companies register to ISO 9001.

Check Your Progress

State whether the following statements are true or false:

1. At the managed level, an organization has developed its own standard software process.
2. With each loop of the spiral, the customer evaluates the work and suggestions are made for its modification.
3. The iterative enhancement model releases software at the end of each cycle.

2.5 LET US SUM UP

Software engineering integrates process, methods and tools for software development. A number of various process models have been proposed each with its own advantages and disadvantages with the underlying generic phases being common.

Attempts like CMM and ISO ensure continuous improvement of the software processes.

2.6 KEYWORDS

CMM: Capability Maturity Model

KPA: Key Process Area

ISO: International Standards Organization

SEI: Software Engineering Institute

SRS: Software Requirement Specification

SDD: Software Design Document

2.7 QUESTIONS FOR DISCUSSION

1. Which of the software development models is the most effective according to you and why?
2. Is Build and Fix Model is an appropriate model for generating a bigger project? Give reasons.
3. While moving outwards in a spiral model, what can be said about the software that is being engineered?
4. Which is more important process or product?

Check Your Progress: Model Answers

1. False
2. True
3. True

2.8 SUGGESTED READINGS

R.S. Pressman, *Software Engineering-A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Rajib Mall, *Fundamentals of Software Engineering*, PHI, 2nd Edition.

Sommerville, *Software Engineering*, Pearson Education, 6th Edition.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

UNIT II

LESSON 3

SOFTWARE PROJECT PLANNING

CONTENTS

3.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Define the software project planning concepts
- Explain estimation of cost, constructive cost model, putnam resource allocation model
- Discuss software risk management

3.1 INTRODUCTION

The project plan documents the planning work necessary to conduct, track and report on the progress of a project. It contains a full description of how the work will be performed.

The project plan includes the:

- Scope and objectives of the project
- Deliverables the project will produce
- Process which shall be employed to produce those deliverables
- Time frame and milestones for the production of the deliverables
- Organization and staffing which will be established
- Responsibilities of those involved
- Work steps to be undertaken
- Budget.

Software Project Planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost and schedule. Software Project Planning

- Understand the scope of the problem
- Make use of past historical data (metrics)
- Perform functional decomposition
- Estimate effort and/or function and/or size
- Perform risk analysis
- Develop a work breakdown structure.

3.2 SOFTWARE PROJECT PLANNING

- Probably the most time-consuming project management activity (or at least it should be).
- Continuous activity from initial concept to system delivery. Plans must be regularly revised as new information becomes available.
- Different types of sub-plans may be developed to support a main software project plan concerned with overall schedule and budget.

Table 3.1: Types of Project Sub-plans

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

```
Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
While project has not been completed or cancelled
loop
        Draw up project schedule
        Initial activities according to schedule
        Wait (for a while)
        Review project progress
        Revise estimates of project parameters
        Update the project schedule
        Re-negotiate project constraints and deliverables
        If (problems arised)
        then
                Initiate technical review and possible revision
        end if
end loop
```

Figure 3.1: Project Planning Process

Project Plan Document Structure

- Introduction (goals, constraints, etc.)
- Project organization
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

Activity Organization

- Activities in a project should be associated with tangible outputs for management to judge progress (i.e., to provide process visibility).
- Milestones are the unequivocal end-points of process activities.
- Deliverables are project results delivered to customers. (There are also internal deliverables.)
- The waterfall model allows for the straightforward definition of milestones ("a deliverable oriented model").
- Deliverables are always milestones, but milestones are not necessarily deliverables.

3.2.1 Milestones in the Re-process

Project Scheduling

- Split project into tasks and estimate time and resources required to complete each.
- Tasks should not be too small or too large they should last on the order of weeks for projects lasting months.
- Organize as concurrent activities to make optimal use of workforce.
- Minimize task dependencies to avoid potential delays.
- Dependent on project managers' intuition and experience.

Process of Scheduling Process

Scheduling Problems

- Estimating the difficulty of problems, and hence the cost of developing solutions, is hard.
- Progress is generally not proportional to the number of people working on a task.
- Adding people to a late project can make it later. (due to coordination overhead)
- The unexpected always happens. Always allow for different contingencies in planning.

Bar Charts and Activity Networks

- Graphical notations are often used to illustrate project schedules.
- Activity charts (a.k.a. PERT* charts) show task dependencies, durations, and the critical path.
- Bar charts (a.k.a. GANTT charts) generally show resource (e.g., people) assignments and calendar time.

*Program Evaluation and Review Technique.

Table 3.2: Task Durations and Dependencies

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

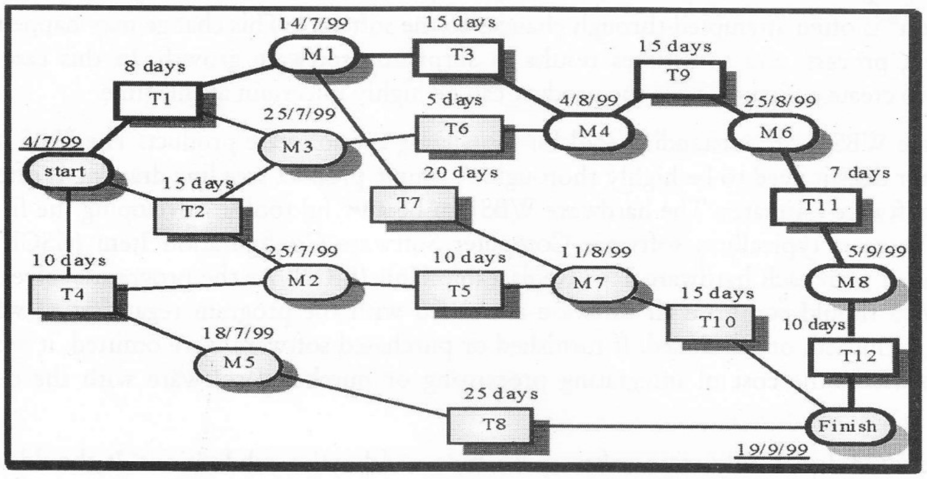

Figure 3.2: Activity Timeline

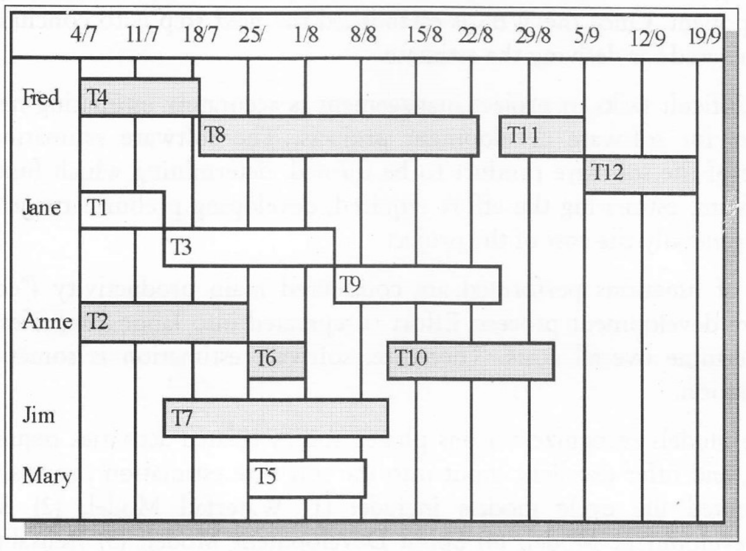

Figure 3.3: Staff Allocation

3.3 COST ESTIMATION

The overall process of raising a cost estimate for software is not different from the process for estimating any other element of cost. There are, however, features of the process that are peculiar to software estimating. Some of the unique aspects of software estimating are determined by the nature of software as a product. Other problems are created by the nature of the estimating methodologies.

Why is it so difficult to estimate the cost of software development? Many of the troubles that plague the development effort itself are responsible for the difficulty come across in estimating that effort. One of the first steps in any estimate is to appreciate and define the system to be estimated. Software, however, is intangible, invisible, and intractable. It is inherently more difficult to recognize and estimate a product or process that cannot be seen and touched. Software produces and changes as it is written. When hardware design has been insufficient, or when hardware fails to perform as expected, the "solution" is often attempted through changes to the software. This change may happen late in the development process, and sometimes results in surprising software growth. In this case it is most important to create a picture, since the product can be highly uncertain at this time.

The software WBS is an outstanding tool for visualizing the software product. The WBS need not be complex, nor does it need to be highly thorough. A simple product tree line drawing is often adequate for initial software estimates. The hardware WBS can be a useful tool in developing the first WBS for software. There is typically a software Computer Software Configuration Item (CSCI) or similar module linked with each hardware Line Replaceable Unit (LRU). As the program evolves, the initial or draft WBS should comprise all software associated with the program regardless of whether it is developed, furnished, or purchased. If furnished or purchased software were omitted, it would not be possible to detain the cost of integrating preexisting or purchased software with the development software.

The WBS should depict only main software functions, and major subdivisions. It should not attempt to describe the software to the hardware it controls. Each of the major software functional units can be modeled as a Computer Software Configuration Item (CSCI). Lower level WBS essentials can be modeled as a component. Once the WBS is established the next step is to conclude which estimating technique should be used for deriving the estimate.

One of the most difficult tasks in project management is accurately estimating needed resources and necessary schedules for software development projects. The software estimation process includes estimating the size of the software product to be formed, determining which functions the software product must perform, estimating the effort required, developing preliminary project schedules, and finally, estimating generally the cost of the project.

Size and number of functions performed are considered main productivity ("complexity") factors during the software development process. Effort is separated into labor categories and multiplied by labor rates to determine overall costs. Therefore, software estimation is sometimes referred to as software cost estimation.

Software life cycle models recognize various phases and associated activities required to develop and maintain software, and offer excellent input into the software estimation process. Some of the more frequent and accepted life cycle models include: (1) Waterfall Model; (2) Rapid Prototyping; (3) Incremental Development Model; (4) Spiral Development Model; (5) Reusable Software Model. These models form a baseline from which to start the software estimation process and should be reviewed and tailored to the planned project.

Moreover, structured approaches to sub-task identification are extremely beneficial in determining tasks and the necessary effort for each task. The WBS is a technique which strongly supports this process.

The software estimation activity should be approached as a chief task and therefore should be well planned, reviewed and continually efficient. The essential steps required to accomplish software estimation are described in the following paragraphs.

Plan the Activities

As before mentioned, the software estimation activity should be planned as a major task. The plan should feature the purpose, products, schedules, responsibilities, procedures, required resources, and assumptions made. The plan should include estimation methodologies, techniques, and tools will be used.

The project should be planned into a hierarchical set of tasks to the lowest level of detail that available information will permit. Additionally, a breakdown of documentation requirements and associated tasks should be defined (the detailed WBS).

The WBS helps to set up a hierarchical view and organization of the project. The top level is the software system or final software product, and subsequent levels help recognize tasks and associated sub-tasks and are used to define and summarize system functions. Each of these tasks is divided into software development phases such as design, code and test, and integration. The entire activities associated with each level must be defined together with: project planning and control, configuration management, product assurance and documentation.

In addition to early growth of detailed knowledge about the project, the WBS provides an excellent methodology for project data collection, tracking, and reporting. During expansion of the project, each of the WBS tasks can be given a project budget, and a job number which is used for reporting time spent on each project phase or activity. This provides an outstanding project tracking and history data collection method. Most government agreements require that such a Cost/Schedule Control System Criteria (C/SCSC) be established. When the data are collected to an established WBS, the information can be placed in a database to be used in refining, tailoring, and modifying the software estimation process. This information becomes a valuable input to the software estimation process for future projects.

Software project tasks/subtasks should be defined to the negligible component potential. All technical aspects of the project should be understood as fully as likely since the more details known about the project the more precise the estimates will be.

3.3.1 Estimation Methodologies

Software cost estimation field has focused on algorithmic cost modeling. In this procedure costs are analyzed using mathematical formulae linking costs or inputs with metrics to create an estimated output. The formulae used in a formal model occur from the analysis of historical data. The accuracy of the model can be improved by calibrating the model to ones detailed development environment, which essentially involves adjusting the weightings of the metrics. There are a variety of different models available, the best known are Boehm's COCOMO, Putman's SLIM, and Albrecht's' function points.

In terms of the estimation process, almost all algorithmic models deviate from the classical view of the cost estimation process.

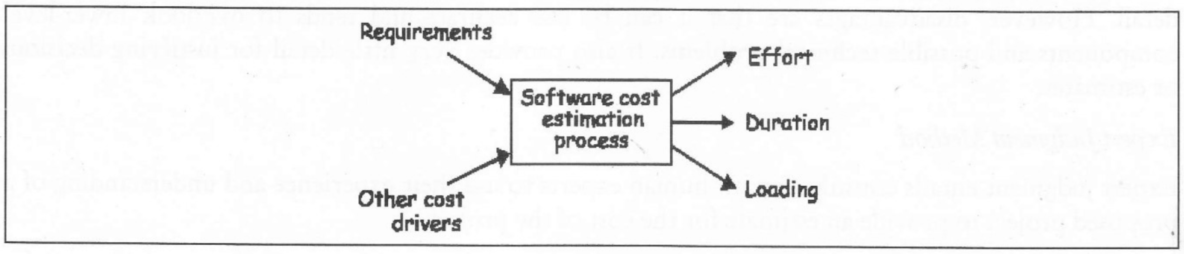

Figure 3.4: Classical View of the Algorithmic Cost Estimation Process

A number of methods (if possible) should be used during the software estimation process. No one methodology is essentially better than the other, in fact, their strengths and weaknesses are often complimentary to each other. It is recommended that more than one software estimation methodology be used for comparison and verification purposes. One method may fail to notice system level activities such as integration, while another method may have included this, but overlooked some key post-processing components. Five of the methods discussed in this lesson are: analogy, bottom-up, top-down, expert judgment, and algorithms (parametric). These methods are frequently used in conjunction with each other and have been used for many years by managers of software projects without the use of any formal software estimation tools. Software estimation tools have only lately been developed which incorporate these methods, and many incorporate multiple methodologies.

Analogy Method

Estimating by analogy means comparing the planned project to previously completed similar projects where project development information is known. Actual data from the completed projects are extrapolated to estimate the planned project. Estimating by analogy can be done either at the system level or the component level.

The main strength of this method is that the estimates are based on definite project data and past experience. Differences between completed projects and the proposed project can be identified and impacts estimated. One difficulty with this method is in identifying those differences. This method is also limited because similar projects may not exist, or the accuracy of available historical data is suspect. The analogy or comparative technique uses parametric approaches including the use of CER's.

Bottom-up Method

Bottom-up estimation entails recognizing and estimating each individual component separately, then combining the results to produce an estimate of the entire project.

It is often complicated to perform a bottom-up estimate early in the life cycle process because the essential information may not be available. This method also tends to be more time consuming and may not be possible when either time or personnel are limited.

Top-down Method

The top-down method of estimation is based on overall characteristics of the software project. The project is partitioned into lower-level components and life cycle phases starts at the highest level. This method is more applicable to early cost estimates when only global properties are known.

Advantages include consideration of system-level activities (integration, documentation, project control, configuration management, etc.), many of which may be unnoticed in other estimating methods. The top-down method is usually faster, easier to implement and requires minimal project detail. However, disadvantages are that it can be less accurate and tends to overlook lower-level components and possible technical problems. It also provides very little detail for justifying decisions or estimates.

Expert Judgment Method

Expert judgment entails consulting with human experts to use their experience and understanding of a proposed project to provide an estimate for the cost of the project.

The obvious advantage of this method is the expert can feature in differences between past project experiences and requirements of the proposed project. The expert can also factor in project impacts caused by new technologies, applications, and languages. Expert judgment forever compliments other estimation methodologies. One disadvantage is that the estimates can be no better than the expertise and judgment of the expert. It is also hard to document the factors used by the expert who adds to the estimate.

Parametric or Algorithmic Method

The algorithmic method engages the use of equations to perform software estimates. The equations are based on research and historical data and use such inputs as Source Lines of Code (SLOC), number of roles to perform, and other cost drivers such as language, design methodology, skill-levels, risk assessments, etc.

Advantages of this method comprise being able to create repeatable results, easily modifying input data, easily refining and customizing formulas, and better understanding of the overall estimating methods since the formulas can be analyzed. However, the results are doubtful when estimating future projects which use new technologies, end equations are generally unable to deal with exceptional conditions such as exceptional workers in any software cost estimating exercises, exceptional teamwork, and an exceptional match between skill-levels and tasks. Nevertheless, any estimating approach can be impacted by these drawbacks, and care should be exercised when accounting for such concerns. Additionally, sometimes algorithms are developed within companies for internal use and many are proprietary as well as more reflective of a specific company's performance characteristics.

An input obligation of an algorithmic model is to provide a metric to measure the size of the finished system. Typically lines of source code are used, this is obviously not known at the start of the project. SLOC is also very dependant on the programming language and programming environment, this is difficult to conclude at an early stage in the problem especially as requirements are likely to be sketchy. Despite this SLOC has been the most widely used size metric in the past, but current trends indicate that it is fast becoming less stable. This is almost certainly due to the changes in software development process in recent years highlighted with a tendency to use prototyping, case tools and so forth. An alternative is to use function points proposed which are related to the functionality of the software rather than its size. A more recent approach is to use object points. This is in contrast a new methodology and has not been publicized in the same depth as function points and SLOC. In essence the method is very similar to function points but counts objects instead of functions. Its recent rise has been encouraged by the interest in the object orientation revolution.

Algorithmic models generally offer direct estimates of effort or duration. Effort prediction models take the general form:

effort = $p \times S$ (1/productivity rate)

where p is a productivity constant and S is the size of the system.

Once the value for p is known. E.g. productivity = 450 source lines of code per month, making $p = 0.0022$ and the size of the system has been estimated at 8500 KLOC.

effort = 0.0022×8500

effort = 18.7 person months

These conclusions indicate that there is greater productivity when building large software systems as opposed to small systems. However, the results can be acceptable as it is expected that larger teams can specialize and the overheads are of a relatively fixed size.

3.4 THE CONSTRUCTIVE COST MODEL (COCOMO)

Constructive Cost Model (COCOMO) is a method for assessing the cost of a software package. COCOMO, Constructive Cost Model is static single-variable model. Barry Boehm introduced COCOMO models. There are three levels of COCOMO model basic, immediate and detailed.

3.4.1 Brief Characteristics of the Model

- *CoCoMo (Constructive Cost Model)* is a combination of parametric estimation equation and weighting method. Based on the estimated instructions (Delivered Source Instructions DSI), the effort is calculated by taking into consideration both the attempted quality and the productivity factors.
- *CoCoMo* is based on the conventional top-down programming and concentrates on the number of instructions.

3.4.2 Levels

- ***Basic CoCoMo:*** Basic COCOMO model is static single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code.By means of parametric estimation equations (differentiated according to the different system types) the development effort and the development duration are calculated on the basis of the estimated DSI. The breakdown to phases is realized in percentages. In this connection it is differentiated according to system types (organic-batch, semidetached-on-line, embedded-real-time) and project sizes (small, intermediate, medium, large, very large).
- ***Intermediate CoCoMo:*** Intermediate COCOMO model computes software development effort as a function of program size and a set of "cost drivers" that include subjective assessments of product, hardware, personnel, and project attributes. The estimation equations are now taking into consideration (apart from DSI) 15 influence factors; these are product attributes (like software reliability, size of the database, complexity), computer attributes (like computing time restriction, main memory restriction), personnel attributes (like programming and application experience, knowledge of the programming language), and project attributes (like software development environment, pressure of development time). The degree of influence can be classified as very low, low, normal, high, very high, extra high; the multipliers can be read from the available tables.
- ***Detailed CoCoMo:*** Advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step, like analysis, design, etc.In this case the breakdown to phases is not realized in percentages but by means of influence factors allocated to the phases. At the same time, it is differentiated according to the three levels of the product hierarchy (module, subsystem, system); product-related influence factors are now taken into consideration in the corresponding estimation equations.

3.4.3 Appraisal of the Model

Applications of CoCoMo

- *Medium and Large Projects:* For small projects, the attempt for an estimation according to intermediate and detailed CoCoMo is too high; but the results from basic CoCoMo alone are not sufficiently exact.
- *Technical Application:* For software projects developing commercial applications, CoCoMo usually comes up with overstated effort estimation values therefore CoCoMo is only applied for the development of technical software. This circumstance is due to the fact that the ratio DSI and man months implemented in the CoCoMo estimation equation fits the efficiency rate in a technical development; with regard to commercial software development a higher productivity rate DSI/man-month can be assumed.

Strong and Weak Points of the Model and possible Remedial Measures

- *Estimation Base "Delivered Source Instructions":* By means of estimation base instructions (DSI) it was attempted to diminish the great uncertainties and problems in connection with the traditional estimation base LOC. However, some problems remain: the ambiguity of a DSI estimation and for the development effort the DSI are-based on modern software engineering methods-no longer of great importance since the effort increasingly occurs during the early activities and DSI will only be effective towards the end of the development process; DSI as well as LOC depends on the selected programming language (an Ada adoption to CoCoMo is already available, however). A remedy can be achieved by the weighting of instructions according to their various types compiler, data description, transformation, control, and I/O instruction, data description instructions (differentiated according to integration degree, message/data object, modification degree) and processing instructions (differentiated according to batch/on-line, modification degree, complexity, language)).
- *Macro and Micro Estimation:* By means of the different levels of the model, CoCoMo makes it possible to realize both a macro estimation by means of Basic CoCoMo and a micro estimation by means of Intermediate CoCoMo and Detailed CoCoMo. The micro estimation allows the effort allocation to activities and functional units. However, method CoCoMo is not only based on a software life cycle deviating from the V-Model but also on another system structure. Therefore, in order to list individual efforts for sub models, (sub-) activities, and (sub-) products, it is necessary to adjust these items of method CoCoMo to the V-Model concept.
- *Influence Factors/Objectivity:* In the effort estimation, CoCoMo takes into consideration the characteristics of the project, the product, and the personnel as well as of the technology. In order to achieve an objective evaluation of these influence factors, CoCoMo offers exact definitions. The quantification of influence factors represents a certain problem, though which has a strong impact on the quality of the estimation method and on the required DSI information.
- *Range of Application:* By differentiating the estimation equations according to project sizes and system types, the range of application for method CoCoMo is a wide one. It is also one of the few estimation methods offering-apart from the support for development projects-support for the effort estimation of SWMM tasks as well (also by parametric estimation equations) as for the estimation of the project duration.

- *Tool Support:* Computer-based support is required for Intermediate and Detailed CoCoMo, based on the quantity problem (differentiation of influence factors on phases and sub products.

3.4.4 Modes

COCOMO can be applied to the following software project's categories.

- *Organic mode:* These projects are very easy and have small team size. The team has a good application experience work to a set of less than inflexible/rigid requirements. A thermal analysis program developed for a heat transfer group is an example of this.
- *Semi-detached mode:* These are intermediate in size and complexity. Here the team has mixed experience to meet up a mix of rigid and less than rigid requirements. A transaction processing system with fixed requirements for terminal hardware and database software is an example of this.
- *Embedded mode:* Software projects that must be developed within a set of tight hardware, software, and operational constraints. For example, flight control software for aircraft.

3.5 THE PUTNAM RESOURCE ALLOCATION MODEL

Staffing Level Estimation

Once the attempt to develop software has been determined, it is necessary to establish the staffing requirement for the project. Putnam first studied the problem of what should be a proper staffing pattern for software projects. He extended the work of Norden who had earlier investigated the staffing pattern of research and development (R&D) type of projects. In order to understand the staffing pattern of software projects, Norden's and Putnam's results must be understood.

Norden's Work

Norden deliberate the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve (as shown in fig. 3.5). Norden represented the Rayleigh curve by the following equation:

$$E = K/t^2_d * t * e^{-t^2 / 2 t^2_d}$$

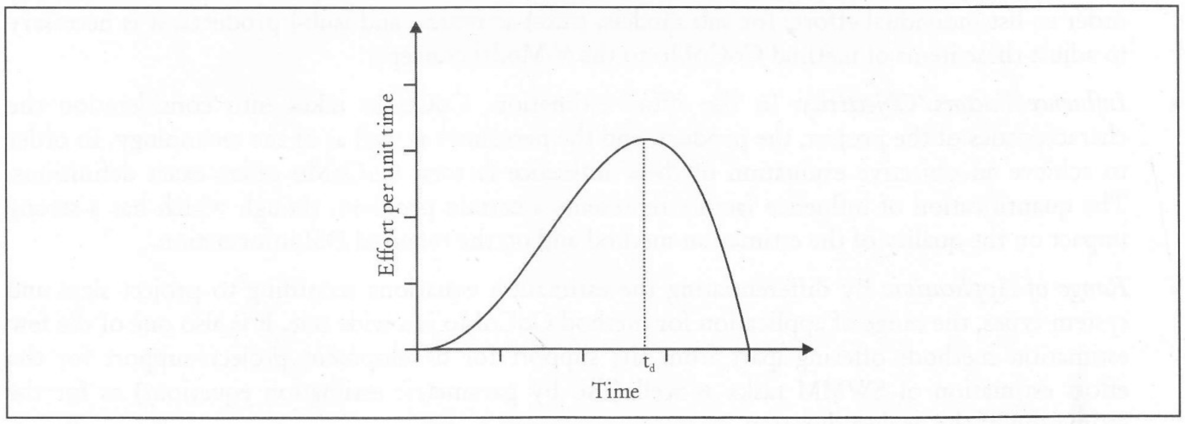

Figure 3.5: Reyleigh Curve

Where E is the effort necessary at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and t_d is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R & D projects and were not meant to model the staffing pattern of software development projects.

Putnam's Work

Putnam deliberate the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration testing. Therefore, t_d can be approximately considered as the time required to develop the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of $C_k = 2$ for poor development environment (no methodology, poor documentation, and review, etc.), $C_k = 8$ for good software development environment (software engineering principles are adhered to), $C_k = 11$ for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from the historical data of the organization developing it.

Putnam recommended that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls.

However, the staff build-up should not be carried out in large installments. The team size should either be increased or decreased slowly when required to match the Rayleigh-Norden curve. Experience shows that a very quick build up of project staff any time during the project development correlates with schedule slippage. It should be clear that a constant level of manpower through out the project duration would lead to wastage of effort and increase the time and effort required to develop the product. If a constant number of engineers are used over all the phases of a project, some phases would be overstaffed and the other phases would be understaffed causing inefficient use of manpower, leading to schedule slippage and increase in cost.

3.6 SOFTWARE RISK MANAGEMENT

What is Risk?

Risk is defined as "The possibility of suffering harm or loss; danger." Even if we're not recognizable with the formal definition, most of us have an innate sense of risk. We are aware of the potential dangers that permeate even simple daily activities, from getting injured when crossing the street to having a heart attack because our cholesterol level is too high. Although we prefer not to dwell on the myriad of hazards that surround us, these risks shape many of our behaviors. Experience (or a parent) has taught us to look both ways before stepping off the curb and most of us at least think twice before ordering a steak. Indeed, we manage personal risks every day.

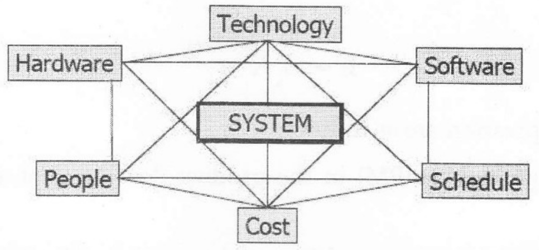

Figure 3.6: Source of Software Risk (System Context)

3.6.1 Description of Risk Analysis

Risk analysis is essentially a "what if" analysis where various scenarios are visualized. It's a systematic use of known information and data to determine how and when incidents can or may occur and the size of their consequences. The management of these risks are a very important part of the management process. This process includes wide aspects of managing and is easiest solved in teams, divided by their skill and knowledge in particular areas. The process of risk analysis and management is a process of continual improvement, which means that there is never only one solution of a problem, there are always improvements that can be done to upgrade (improve) the quality of the treatment of the risks.

3.6.2 Purpose of Risk Management

The risk management process includes seven steps that has to be followed:

Figure 3.7 shows what the concept continues improvements mean. When the risk has been treated it is not put a side, but instead the risk management process starts all over again to come up with better solutions. The seven steps can be explained as the following:

1. Establish the context sets the boundaries for which within the risk are managed and helps to set guidelines for how to get started with the process. The context includes five sublevels:
   (a) The strategic context
   (b) The organizational context
   (c) The risk management context

(d) Develop criteria

(e) Decide the structure

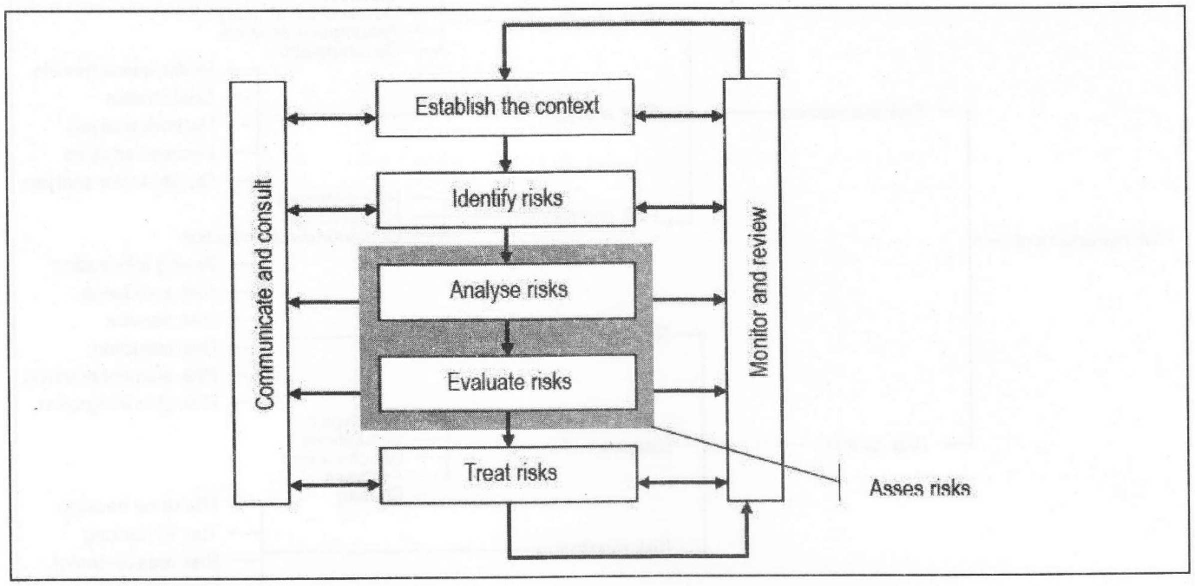

Figure 3.7

2. Identifying risks is the same as identifying how, what or why incidents may occur.

3. The risk analysis includes the magnitude of consequences and the chance that these consequences may come to life.

4. In the evaluation of risks the consequences are leveled (ranked) after their magnitude, so if needed the right treatment will be applied.

5. If the risk is low or has a low-priority it can be taken care of (by routine knowledge) or be accepted at this stage. But if it is a high-priority risk, a plan for managing is instantly laid out. According to AS/NZS 4360 the following plan should be used:

    - Identify treatment options
    - Evaluate treatment options
    - Select treatment options
    - Prepare treatment plan
    - Implement plans

6. Monitoring and reviewing are used to overlook the risk management cycle and track changes within it so new contexts (continues improvements) can be made and the final treatment of the risks improves.

7. Communication and consultation at every new step in the process is very important to people on the inside of the company as for people on the outside (stakeholders, investors).