UNIT – IV

OBJECTIVES OF SOFTWARE QUALITY MANAGEMENT

- To introduce the quality management process and key quality management activities.
- To explain the role of standards in quality management.
- To explain the concept of a software metric predictor (analyst) metrics and control metrics.
- To explain how measurement may be used in assessing software quality and the limitation of software measurement.

Traditionally, a quality product is defined in terms of its fitness of purpose i.e. a quality product does exactly what the users want it to do. For software product fitness of purpose is usually interpreted in terms of satisfaction of requirements laid down in the SRS documents.
"Fitness Of Purpose" is not a wholly satisfactory definition of quality. For example, A software product that performs all function as specified in the SRS documents. But has an almost unusable user interface. Even though the software may be functionally correct. We cannot consider it to be a quality product.

The quality factors with a software product 4) Correctness
Software quality management [Detail explanation will be refer in UNIT – I page no. 19 and 20].
Concerned with ensuring that the required level of quality is achieved in a software product. Involves defining appropriate quality standards and procedures and ensuring that these are followed shoaled aim to develop a "quality culture" where quality is seen as everyone's responsibility.

Software quality management

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

Managerial Structure & Individual Responsibilities

A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management.

Quality System Activities

The quality system activities encompass following –

- Auditing of Projects
- Review of the quality system
- Development of standers, procedures and guidelines etc.
- Production of reports for the top management, summarizing the effectiveness of the quality system in the organization.

A good quality system must be well documented. Without a properly documented quality system, the application of quality controls and procedures becomes adhoc, resulting in large variation in the quality of the products delivered.

Evolution of Quality System

Quality systems have rapidly evolved over the five decades. Quality control aims at correcting the cause of errors and not just rejecting the defecting products. The next breakthrough in

quality system was the development of quality assurance principles. The basic premise of modern quality assurance is that if an organization's process are good and are followed rigorously then the products are bound to be of good quality.

What is quality?

Quality simplistically, means that a product should meet its specification.

This is problematical for software systems

- There is a tension between customer quality requirements (efficiency, reliability, etc.)
- Some quality requirements are difficult to specify in an unambiguous way.
- Software specifications are usually incomplete and often inconsistent.

The quality compromise

We cannot wait for specifications to improve before paying attention to quality management. We must put quality management procedures into place to improve quality in spite of I m perfect specification.

Scope of quality management

Quality management is particularly important for large, Complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes, for smaller systems, quality management needs less documentation and should focus on establishing a quality culture.

Quality management activities

- Quality assurance: Establish organizational procedures and standard for quality.
- Quality planning: Select applicable procedures and standards for a particular project and standards.
- Quality management should be separate from project management to ensure independence.

Process and product quality

-There quality of a develop product is influenced by the quality of the production process.
-This is important is a very complex and poorly understood relationship between software processes and product quality.

Quality assurance and standards

Standards are the key to effective quality management. They may be international, national and organizational or project standards. Product standards define characteristics that all components should exhibit e.g. common programming style. Process standards define how the software process should be enacted.

Importance of Standards

Encapsulation of best practice – avoids repetition of past mistakes. They are a framework for quality assurance processes – they involve checking compliance to standers. They provide

continuity – new staff can understand the organization by Understanding the standards that are used.

Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Requirement document structure	Submission of documents of CM
Method header format	Version release process
Java programming style	Project plan approval process
Project format	Change control process
Change request form	Test recording process

Problems with standards

Software engineers may not see them as relevant and up-to-date. They often involve too much bureaucratic from filling. If they are unsupported by software tools, tedious manual work is often involved to maintain the documentation associated with the standards.

Standards development

Involve practitioners in development. Engineers should understand the rationale Underlying a standard. Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners. Detailed standards should have associated tool support. Excessive clerical work is the most significant complaint against standards.

Quality Planning

A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes. The quality plan should define the quality assessment process. It should set out which organizational standards should be applied and where necessary. Define new standards to be used.

Quality Plans

- Quality plan structure
- Product introduction
- Product plans
- Process descriptions
- Quality goals

Risks and risk management.

Software quality attributes

[Note : answer in UNIT – I page no : 19].

Quality control

This involves checking the software development process to ensure that procedures and standards are being followed. There are two approaches to quality control quality reviews : Automated software assessment and software measurement.

Quality reviews

This is the principle method of validating the quality of a process or of a product.

A group examines part of all of a process or system and its documentation to find potential problems. There are different types of review with different objectives.

- Inspections for defect removal(product).
- Reviews for progress assessment (product and process).
- Quality reviews (product and standards).

Types of review

Review type	Principle Purpose
Design or program inspection	To detect detailed errors in the requirements, design or code. A checklist of possible errors should drive the review.
Progress review :	To provide information for management about the overall progress of the project. This is both a process and product review and is concerned with costs, plans and schedule.
Quality review :	To carry out a technical analysis of product components or documentation to find mismatches between the specification and the component design. Code or documentation and to ensure that defined quality standards have been followed .

QUE : EXPLAIN SOFTWARE MEASUREMENT AND METRICS

Software measurement is concerned with deriving a numeric value for an attribute of a software product or process. This allows for objective comparisons between techniques and processes. Although some companies have introduced measurement programs, most organization still don't make systematic use of software measurement.

There are few established standards in this area.

Software metric

Any type of measurement, which relates to software system, process or related.

Documentation

- Lines of code in a program, the Fog index, and number of person-day required to develop a component.

- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process. Product metrics can be used for general predictions or to identify anomalous components.

Predictor and control metrics

Metrics assumptions

A software property can be measured. The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but often more interested in external software attributes.

This relationship has been formalized and validated. It may be difficult to relate what can be measured to desirable external quality attributes.

Internal and external attributes

The measurement process

A software measurement process may be part of a quality control process. Data collected during this process should be maintained as an organizational resource. Once a measurement database has been established, comparisons across projects become possible.

Product measurement process

Data collection

A metrics programmer should be based on a set of product and process data. Data should be collected immediately (not in retrospect) and, if possible, automatically. Three type of automatic data collection.

- Static product analysis
- Dynamic product analysis
- Process data collection.

Data accuracy

Don't collect unnecessary data the questions to be answered should be decided in advance and the required data identified. Tell people why the data is being collected. It should not be part of personnel evaluation. Don't rely on memory collected data when it is generated not after a project has finished.

8.6.2 Product metrics

A quality metric should be a predictor or product quality. Classes of product metric are :

Dynamic and static metrics:

Dynamic metrics are closely related to software quality attributes, it is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).

Static metrics have an indirect relationship with quality attributes You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

8.6.4 Complexity Metrics

Prominent in the history of software metrics has been the search for measures of complexity. This search has been inspired primarily for the reasons discussed above (as a necessary component of size) but also for separate QA purposes (the belief that only be measuring complexity can we truly understand and conquer it.) Because it is a high-level nation made up to many different attributes, there can never be a single measure of software complexity metrics. Most of these are also restricted to code. The best known are Halstead's software science and McCabe's cyclomatic number.

Q. QUE : EXPLAIN SEI CAPABILITY MATURING MODEL (CMM)

SEI capability maturity model was proposed of the Software Engineering Institute of the Carnegie Mellon University. USA . The SEI CMM was originally developed to assist the US development of defense in acquisition of software. The SEI CMM helped organization to improve quality of their software and therefore adoption of SEI CMM has significant business benefits.

Capability Maturity Model (short, names, CMM, PCMM) is a collection of instructions an organization can follow with the purpose to again better control over its software development process. The CMM ranks software development organizations according to a hierarchy of five process maturity levels. Each level ranks the development environment according to its capability of producing quality software. A set of standards is associated with each of the five levels. The standards for level one describe the most immature, or chaotic, process, and the standards for level five describe the most mature. Or quality, process. Currently an estimated 75% of software development organizations achieve only level 1 standards.

The Capability Maturity Model (CMM) is a way to develop and refine an organization's software development process. A maturity model is a structured collection of elements that describe characteristics of effective processes. A maturity model provides :

- A place to start
- The benefit of a community's prior experiences
- A common language and a shared vision
- A framework for prioritizing actions
- A way to define what improvement means for your organizations

A maturity model can be used as a benchmark for assessing different organizations for equivalent comparison.

Levels of the CMM

The SEI has subsequently released a revised version known as the Capability Maturity Model Integration (CMM). SEI, CMM is a reference model which classifies the software development industry into the following five maturity levels.

Level 1 – Initial

At maturity level 1, processes are usually ad hoc and the organization usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization and not an the use of proven processes. In spite of this ad hoc. Chaotic environment, maturity level 1 organizations often produce and services that work, however, they frequently exceed the budget and schedule of their projects.

Maturity level 1 organizations are characterized by a tendency to over commit abandon processes in the time of crisis, and not be able to repeat their past successes again.

Very few or no process are defined and followed.

Level 2 – Repeatable

At maturity level 2, software development successes are repeatable. The organizations may use some basic project management to track cost and schedule.

Process discipline helps ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

Project status and the delivery of services are visible to management at defined points (for example, at major milestones and at the completion of major tasks).

Basic project management processes are established to track cost, Schedule, and functionality. The minimum process discipline is in place to repeat earlier successes on projects with similar applications and scope. There is still a significant risk of exceeding cost and time estimate.

Level 3 – Defined

The organization's set of standard processes, which is the basis for level 3, is established and improved over time. These standard processes are used to establish consistency across the organization.

Projects establish their defined processes by the organization's set of standard processes according to tailoring guidelines. The organization's management establishes process objectives based on the organization's set of standard processes and ensures that these objective are appropriately addressed.

A critical distinction between level 2 and 3 is the scope of standards, process descriptions. And procedures. At level 2, the standards, process descriptions, and procedures may be quite different each specific instance of the process (for example, on a particular project). At level 3,

the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit.

Level 4 – Managed

Using precise measurements, management can effectively control the software development effort. In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications.

Sub processes are selected that significantly contribute to overall process performance. These selected sub processes are controlled using statistical and other quantitative techniques.

A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.

Level 5 – Optimizing

Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements. Quantitative process improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process- improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.

Process improvements to address common causes of process variation and measurably improve the organization's process are identified, evaluated, and deployed. Optimizing processes that are nimble, adaptable and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization's sbility to rapidly respond to changes and opportunities is enhances by finding ways to accelerate and share learning.

A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process (that is, shifting the mean of the process performance) to improve process performance (while maintaining statistical probability) to achieve the established quantitative process- improvement objectives.

Within each of these maturity levels are KP As (key Process Areas) which characterize that level, and for each KPA there are five definitions identified:

1. Goals
2. Commitment
3. Ability

4. Measurement
5. Verification

Sr.	CMM Level	Focus	Key Process Area
1.	Initial	Competent People	
2.	Repeatable	Project management	Software Project Planning Software configuration Management
3.	Defined	Definition of Process	Process definition Training programme peer reviews
4.	Managed	Product and Process Quality	Quantitative process metrics, Software quality management
5.	Optimizing	Continuous Process Improvement	Defect prevention process change management.

Note :- The CMM was originally intended as a tool to evaluate the ability of government contractors to
Perform a contracted software project. It may be suited for that purpose. When it become a general model for software process improvement. There were many critics.

COMPUTER AIDED SOFTWARE ENGINEERING (CASE) AND IT'S SCOPE

CASE tools helps to reduce the efforts on software development and maintenance as well as reduction in development and maintenance costs.

Case tool is a generic term used to denote any from of automated support for software engineering and used to automate some activities associated with software development. Some of these tools assist in phase-related tasks such as specification, structured analysis, design, coding, testing etc. and others in non-phase activities such as project management and configuration management.

The primary objectives of developing or using any CASE tools are :

- To increase productivity
- To help produce better quality software at lower costs.

Benefits of CASE

1) Cost saving through all development phases
2) Improvement to the quality
3) Help to produce high quality and consistent documents
4) Reduce the effort in Software Engineering work.
5) Have led to revolutionary reduction in software maintenance costs.
6) It has an impact on the style of working of company.

CASE Support is Software Life Cycle

The various types of support that case provides during the different of a software life cycle.

1) Prototyping Support

2) Structured Analysis and Design

3) Code Generation

4) Test Case Generator

OTHER CHARACTERISTICS OF CASE TOOLS

The definition of objects should be organized graphically and able to incorporate text and diagrams from the control repository. This helps in producing up to date documentation.

The CASE tools should integrate with one or more of the commercially available desk-top publishing packager. It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as postscript.

Project management

The CASE tool should support the task of collecting, storing and analyzing information on the software project progress such as estimated task duration, scheduled and actual task start, complication data, dates and results of reviews etc.

External Interface

The CASE tools should allowed exchange of information for reliability of design. The information , which is to be exported by the tools should be preferably in ASCII format and support open architecture. similarly the data dictionary should be provide programming interface to access information. It is required for integration of custom utilities, building new techniques or populating data dictionary

Reverse Software Engineering

The CASE tools should support generation of structure charts and data dictionary from the existing source code. It should populate the data dictionary from the source code. If the tools is used for re-engineering the information system. It Should contains conversion tools from indexed sequential file structure, hierarchical and network database to RDBMS.

data dictionary interface

[PNO : 74] ARCHITECTURE OF CASE ENVIRONMENT

SOFTWARE RELIABILITY AND QUALITY ASSURANCE

SOFTWERE RELIABILITY

Reliability of a software product is an important concern for most users. Users not only want highly reliable products, but also want quantitative estimation of the reliability of a product before taking a buying decision. However, it is very difficult to accurately measure the reliability of any software product. One major problem is that there are no good metrics available, using which we can quantify the reliability of a product.

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

The reliability of a system would improve if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent software defects in the system.

It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent on executing only 10% of the instruction in the program. These most used 10% instruction are called core part of the program.

The rest 90% of the program statements are called non-core and are executed only 10% of the total execution time. It is clear that removing 60% defects from the least used parts of system would lead to only 3% improvement in reliability. It is thus clear that improvement in the overall reliability of a program due to the correction of a single error would depend on whether the error belonged to the core part or the non-core part of the program.

The reliability also depends on location of the program It also depends on how the product is used i.e. its execution profile. So the reliability figure of a software product is clearly observer dependent and cannot be determined correctly.

RELIBILITY METRICS

Since different categories of software products have different reliability requirements. It is necessary that the levels of reliability be specified in the SRS documents. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product. A good reliability measures should be observer- dependent so that different people can agree on the degree of reliability that a system has.

In practice, it is very difficult to formulate a precise technique for measuring reliability. The next best thing is to have measures that correlate with reliability. For example the number of well designed test cases a system processes correlates with its correctness. The six reliability metrics are used to quantify the reliability of software products.

1) Rate of Occurrence of Failure(ROCOF)

Measures the frequency of occurrence of unexpected behaviors (i.e. failures). A ROCOF of 2/100 means that two failures are likely to occur in each 100 operational time units. This metrics is sometimes called the failure intensity. This metric should be used where regular demands are made son system services and where it is important that these services are correctly delivered. Ex. Bank teller system.

2) Mean Time to Failure(MTTF)

The average time between observed system failures. An MTTF of 500 means that one failure can be expected every 500-time units. This metric should be used in system where there are long transaction, i.e. where people use the system for a long time. The mean time to failure should be longer than the average length of transaction. Ex. Word processor system.

3) Availability

The probability that the system is available for use at a given time Availability 0.0998 means that in every 1000 time units, the system is likely to be available for 998 of these. This metric should be used in non-stop system where users expect the system to deliver a continuous service. Ex. Railway signaling system.

4) Mean Time to Repair(MTTR)

Once failure occurs. Some time is required to fix the error. MTTR measures the ever age time it takes to track the errors causing the failure and then to fix them.

5) Probability of failure on demand (POFOD)

This metric is most appropriate for system where services are demanded at unpredictable or at relatively long time intervals and where there are serious consequences if the service is not delivered. A POFOD of 0.001 means that one out of a thousand service request may result in failure. Ex. Emergency shutdown system in a power plant.

6) Mean Time Between Failures (MTBF0

MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected to occur only after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF.

All the above reliability metrics are centered around the probability of a system failures but takes no account of the consequences of failures. In other words, these reliability models take no account of the relative severity of different failures.

In order to study, the reliability of a software product, it is therefore necessary to classify failures into their various types.

A possible classification of failure is follows :

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

2 Permanent – Permanent failures occurs for all input values while invoking a function of the system.

3 Recoverable – When recoverable failures occurs. The system recovers with or without operator intervention.

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

5 Cosmetic – Such Failure can causes minor irritants, but do not lead to incorrect results. An example of a cosmetic failure may be the case where the mouse button has to be clicked twice instead of once to invoke a given function through graphical user interface.

RELIBILITY GROWTH MODELING

A reliability growth model is a mathematical model of how software reliability grows as errors are detected and repaired. A reliability growth model can be used to predict when a particular level of reliability is likely to be attained, Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. There are several different reliability growth models have been proposed, among those following two are mostly used.

Step Function Model (Helsinki and Miranda Model)

The simplest reliability growth model is a step function model, where it is assumed that reliability increases by constant increment each time one error is detected and repaired. This model assumes that software repairs are always correctly implemented so that the number of software faults and associated failure decreases with time. As repair are made. The rate of occurrences of software failures (ROCOF) should decrease.

Little wood and Verrall Model

In this model, it is realized that reliability doest not increase by a constant amount each time on error is repaired. The improvement in reliability due to fixing of an error is assumed to do proportional to the number of errors remaining in the system at that time. Therefore, the growth of reliability is not constant in each time increment. They took a random element into the reliability growth improvement effected by a software repair. Thus, each repair does not result in little wood and Verrall's Model allow for negative reliability growth when a software repair introduces further errors.

ISO-9000 CERTIFICATION FOR SOFTWARE INDUSTRY

An international set of standards for quality management. Application to a range of organization from manufacturing to service industries. ISO 9001 application to organizations. Which design,

develops and maintains products. ISO 9001 is a generic model of the quality process that must be instantiated for each organization using the standard.

ISO 9001

Management responsibility	Quality system
Control of non-conforming products	Design control
Handling storage. packaging & delivery	Purchasing
Purchaser-supplied products	Product identification & traceability
Process control	Inspection &testing
Inspection 7 test equipment	Inspection & test status
Contract review	Corrective action
Document control	Quality records
Internal quality audits	Training
Servicing	Statistical techniques

ISO 9000 CERTIFICATION

Quality standards and procedures should be documented in an organization quality manual. An external body may certify that an organization's quality manual conformance to ISO 9000 standards. Some customers require supplies to be ISO 9000 certified although the need for flexibility here is increasingly recognized.

ISO 9000 and quality management

Certification for software Industry

International standards organization is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987. The ISO 9000 certification serves as a reference for contract between independent parties. The ISO 9000 standard specifies the guidelines for maintaining a quality system ISO 9000 is a series of three standards.

ISO 9001

This standard applies to the organizations engaged in design development, production and serving of good. This is the standard that is applicable to most software development organizations.

ISO 9002

This standard applies to the organization that do not design products but are only involved in production. Example - steel or car manufacturing industries.

ISO 9003

This standard applies to organizations involved only in installation and testing of the products. ISO 9000 is a generic standard that is applicable to a large amount of industries, ranging from a steel manufacturing industry to a service rendering company. Therefore, many of the clauses of

the ISO 9000 documents are writing using generic terminologies and it is very difficult to interpret them in the contest of software development organization. There are two primary reasons underlying this :

1) Software is intangible and therefore difficult to control
2) During the development of software product the only raw material consumed is data, whereas large quantities of raw materials in physical forms are consumed for producing all other products.

Due to such radical differences between software and other products it was difficult to interpret various clauses of the original ISO standards in the context of software industry.

8.5.4 Why Get ISO Certification?

There is a mad scramble among software development organization for obtaining the ISO certification due to the benefits it offers.

1) Confidence of customers in an organization is enhanced when organization has the ISO 9001 certification.
2) ISO 9000 requires a well-documented software production process contributes to repeatable and higher quality of the developed software.
3) ISO 9000 makes the development process focused, efficient and cost effective .
4) ISO 9000 points out the weak points of an organization and recommends remedial action.
5) ISO 9000 sets the basic framework for development of an optimal process and TQM.

Documentation standards
Particularly important- documents are the tangible manifestation of the software.
Documentation process standards Concerned with how documents should be developed validated and maintained. Document interchange standards Concerned with the compatibility of electronic documents.

Documents standards

- Documents identification standards how documents are uniquely identified.
- Documents structure standards Standard structure for project documents.
- Documents presentation standards Define fonts and styles, use of logos, etc.
- Documents update standards Define how changes from previous versions are reflected in a document.

Documents interchange standards
Interchange standards allow electronic documents to be exchanged. mailed, etc. Documents are produced using different system and on different computers. Even when standards tools are used, standards are needed to define convention for their use e.g. use of style sheets and macros.

Need for archiving. The lifetime of word processing system may be much less than the lifetime of the software being documented. An archiving standard may be defined to ensure that the documents can be accessed in future.

METRICS OF FUNCTIONALITY :

ALBRECHT'S FUNCTION POINTS

The COCOMO type approach to resource estimation has two major drawbacks, both concerned with its key size factor KDSI :
KDSI is not known at the when estimations are sought, and so it also must be predicted.
This means that we are replacing one difficult prediction problem (resource estimation) with another which may equally as difficult (size estimation).

KDSI is a measure of length. Not size (it takes no account of functionality or complexity)

Albrecht' s Function Points (FPs) is a popular product size metric (used extensively in the USA and Europe) that attempts to resolve these problems. FPs are supposed to reflect the user's view of a system's functionality. The major benefit of FPs over the length and complexity metrics discussed above is that they are not restricted to code. In fact they are normally computed from a detailed system specification, using the equation.

FP = UFC ? TCF

Where UFC is the Unadjusted (or Raw) Function Count, and TCF is a Technical Complexity Factor, which lies between 0.65 and 1.35. The UFC is obtained summing weighted counts of the number of inputs, outputs, logical master files, interface files and queries visible to the system user, where :

- An input is a user or control data element entering an application.
- An output is a user or control data element leaving an application.
- A logical master file is a logical data store acted on by the application user.
- An interface file is a file or input/output data that is used by another application.
- A query is an input- output combination (i.e. an input that results in an immed data output).

The weights applied to simple, average and complex elements depend on the element type. Elements are assessed for complexity according to the number determined by rating the importance of 14 factors on the system in question. Organizations such as the international Function Point Users Group have been active in identifying rule for function point counting to ensure that count are comparable across different organizations.

Function points are used extensively as a size metric in preference to LOC. Thus, for example, they are used to replace. LOC in the equations for productivity and defect density. There are

some obvious benefits: FPs are language independent and they can the computed early in a project. FPs are also being used increasingly in new software development contacts.

One of the original motivations for FPs was as the size parameter for effort prediction. Using FPs avoids the key problem identified above for COCOMO : we do not have to predict FPs : they are derived directly from the specification which is normally the documents on which we wish to base out resource estimates.

The major criticism of FPs is that they are unnecessarily complex. Indeed empirical studies have suggested that the TCF adds very little in practical terms. For example effort prediction. Using the unadjusted function count is often no worse then when the TCF is added. FPs are also difficult to compute and contain a large of subjectivity. There is also doubt they do actually measure functionality.

Measurement analysis

It is not always obvious what data means Analyzing collected data is very difficult. Professional statisticians should be consulted if available. Data analysis must take local circumstances into account.

Measurement surprises

Reducing the number of faults in a program leads to an increased number of help desk calls The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase. Amore reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

Key points

Software quality management is concerned with ensuring that software meets its required standards. Quality assurance procedures should be documented in an organizational quality manual. Software standard are an encapsulation of best practice. Reviews are the most widely used approach for assessing software quality. Software measurement gathers information about both the software product. Product quality metrics should be used to identify potentially problematical components. There are no standardized and universally applicable software metrics.

Comparison between ISO and SEI CMM

1) ISO 9000 is awarded by an international standards body, therefore ISO 9000 certification can be quoted by an organization in official documents, in communication with external parties, and in tender quotations while SEI CMM assessment is purely for internal use.

2) SEI CMM was developed specially for software industry and therefore addresses many issues, which are specific to software industry alone.

3) SEI CMM goes beyond quality assurance and aims at TQM. In fact ISO 9001 aims at level 3 of SEI CMM.

4) SEI CMM provides a list of key process areas (KPAs) on which an organization at any maturity level needs to concentrate to take if from one maturity level to the next. Thus. It provides a way for achieving quality improvement gradually.

Software re – engineering

Legacy systems are old software systems which are essential for business process support. Companies rely on these systems so they must keep them in operation. Software evolution strategies include the maintenance, replacement architecture evolution and software re-engineering. Reorganizing and modifying existing software system to make them more maintainable. Re-engineering may involve re-documenting organizing the system organizing and restructuring the system, translating the system to more modern programming the language and modify the updating structure and value of the systems data. Re-sturcturing or rewriting part of the all of a legacy system without changing and its functionality.

Software re-engineering applicable where some but not all sub-system of a larger system require frequent maintenance, Re-engineering involves adding effort to make them easier to maintain. the system may be re-structured and re-documented.

When-to-re-engineer:

- When system changes are mostly confined to part of the system then re-engineering their part.
- When hardware or software support becomes obsolete.
- When tools to support re-structuring are available.

- CASE tool is a generic term used to denote any form of automated support for software engineering and used to automate some activities associated with software development.
- Some of these tools help in phase-related tasks such as specification, structured analysis, design, coding, testing etc. and others in non-phase activities such as project management and configuration management.
- The primary objective of developing or using any CASE tools are :
  - **To increase productivity**
  - **To help produce better quality software at lower costs**
- The true power of tool set can realized only when tools are integrated into a common framework or environment.

Benefits of CASE

(i) Cost saving through all development phases
(ii) Improvement to the quality
(iii) Help to produce high quality and consistent document
(iv) Reduce the effort in Software Engineering work.
(v) Have led to revolutionary reduction in software maintenance costs.
(vi) It has an impact on the style of working of company

⌘ BUSINESS PROCESS RE-ENGINEERING

- **Business process re-engineering is concerned with re-designing business processes to make them more responsive and more efficient.**
- It is usually reliant on the introduction of new computer systems to support the revised processes.
- Process re-engineering is often a driver for software evolution as legacy systems may incorporate implicit dependencies on the existing processes.
- The need for software re-engineering may surface in a company when it becomes clear that the scale of the changes required by the business process re-engineering cannot be accommodated through normal program maintenance.
- The critical distinction between re-engineering and new software development is the starting point for the development.
- Rather than start with a written specification, the old system acts as a specification for the new system.

Re-engineering process

- The input to the process is legacy program and output is structured, modularized version of the same program.
- At the same time as program re-engineering, the data for the system may also be re-engineered.
- The activities in this re engineering process are:
  - *Source code translation*: Source code gets converted from an old language to new programming language.
  - *Reverse engineering*: The program is analysed and information extracted from it. which helps to document its organization and functionality.
  - *Program modularization*: The control structure of the program is analyzed and modified to make it easier to read and understand.
  - *Program modularization*: The process of re-organizing a program so that related program parts are collected together in a single module.
  - *Data re-engineering*: The data processed by the program is changed to reflect program changes.

અશ્વિન - મો.૯૫૮૪૧૮૮૬૮૬
અલ્કેશ - મો.૯૬૭૪૫૪૦૩૮૮
[illegible]
[illegible] ટ્રાન્સપોર્ટની સામે, ગંજબજાર,
રેલ્વે ફાટક, વિસનગર.

SOFTWARE REVERSE ENGINEERING

- Software Reverse engineering is the process of recovering the design and requirement specification of a product from an analysis of its code.
- The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system (old system).
- Reverse engineering is become important, since several existing software products lack proper documentation, are highly unstructured or their structure has degraded through a series of maintenance efforts.
- The first stage in reverse engineering is usually to do cosmetic changes to the code to improve its readability, structure and understandability without changing any of its functionalities.
- Reforming a program can be achieved by using several of the available petty printer programs which layout the program neatly.
- The different variable, data structure and functions should be assigned meaningful names wherever possible.

- Complex nested conditional in the replaced by simpler conditional statements or whenever appropriate by case statements.

WHAT IS RISK?

- A simple definition of a "risk" is a problem that could cause some loss or intimidate the success of our project, but which hasn't happened yet. (And we'd like to keep it that way.)
- These potential problems might have an adverse impact on the cost, schedule, or technical success of the project, the quality or our software products, or project team morale.
- Risk management is the process of identifying, addressing, and eliminating these potential problems before they can damage our project.
- We need to differentiate risks, as potential problems, from the current problems facing the project, because different approaches are taken for addressing these two kinds of issues.
- For example, a staff shortage because you haven't been able to appoint people with the right technical skills is a current problem, but the danger of your top technical people being appoint away by the competition is a risk.
- Current, real problems require prompt corrective action, whereas looming risks can be dealt with in several different ways.
- We might choose to avoid the risk entirely by changing our project approach or even canceling the project. Or, we could elect to simply attract the risk and take no specific actions to avoid or minimize it.

WHY MANAGE RISKS FORMALLY?

- A formal risk management process provides a number of benefits to the project team. First, it gives us a structured mechanism to provide visibility into threats to project success.
- By considering the potential impact of each risk item, we can make sure we focus on controlling the most severe risks first.
- A team approach allows all various project stakeholders to collaboratively address their shared risks, and to assign responsibility for risk improvement to the most appropriate individuals.
- We can combine risk assessment with project estimation to quantify possible schedule slippage if certain risks materialize into problems.
- Without a formal approach, we cannot ensure that our risk management actions will be initiated in a timely fashion, completed as planned, and effective.
- The net result of these activities is to help avoid preventable surprises late in the project, and therefore improve the chance of meeting our commitments.

- The entire development organization also enjoys benefits from risk management. Sharing what does and does not work to control risks across multiple projects helps projects avoid repeating the mistakes of the past.
- Members of the organization can pool their experience and identify opportunities to control our most common risks, through education, process improvement, and application of improved software engineering and management techniques.

⌘ RISKS AND UNCERTAINTY (Insecurity)

- Much risk is attributable to uncertainty about the things we sometimes pretend are under control.
- It's what we don't know that can hurt us.
- Uncertainty is a normal and necessary characteristic of most software projects.
- It can result from the continuously increasing complexity of the products we create. and the haste with which we sometimes dive into the source code editor.
- When you're living on the bleeding edge of rapidly changing technology or business conditions, uncertainty permeates your life.
- Lack of practical knowledge about the software development techniques and tools we are using presents an additional source of uncertainty.
- Controlling risk partly means reducing uncertainty.
- Of course, reducing uncertainty has a cost.
- We need to balance such costs against the potential cost we could incur if the risk is not addressed and does indeed bite us.
- It may not be cost-effective to reduce uncertainty too much.
- For example, if we are concerned about the ability of a subcontractor to deliver an essential component of our product on time, we could engage multiple subcontractors to increase the likelihood that at least one will come through on schedule.

⌘ TYPICAL SOFTWARE RISKS

- The list of wicked things that can happen a software project is depressingly long.
- The open-minded project manager will acquire extensive lists of these risk categories to help the team uncover as many concerns as possible early in the planning process.
- Possible risks to consider can come from group think activities, or from a risk factor chart accumulated from previous projects.
- In one group I've worked in, individuals came up with very insightful similes of their risk factors, which I edited together and we then reviewed as a team.
- The software Engineering Institute has assembled classification of hierarchically-organized risks in 13 major categories, with about 200 thought-provoking questions to help you spot the risks facing your project.

RISK MANAGEMENT AND SCHEDULE ESTIMATION

- You may have had the experience of preparing an estimate for some body of work, only to have it cut in half by a skeptical manager.
- Similarly, arbitrary "cheat factors" to account for the possibility of unknown events may be discarded by managers who always expect the best possible outcome.
- Quantitative risk management can build the case for contingency buffers that are more likely to be taken seriously by the decision makers.
- When estimating the potential loss associated with each risk item, consider the impact a materialized risk could have on your schedule.
- Consider a project dependency on having a necessary component delivered from a subcontractor on schedule.
- You estimate the potential schedule loss if this delivery doesn't happen as 4 weeks, and the probability as 0.3 The risk exposure from this item is therefore 4 weeks times 0.3, or 1.2 weeks.
- By performing a similar analysis for each of your top 10 risk factors, you can total the individual risk exposures to estimate the cumulative risk exposure from those items, in units of calendar time (or, possibly, actual dollars), as shown below.

Total risk exposure = Sum of (probability X Impact)

- You can't know exactly which of the identified risk factors might turn into problems.
- However, by quantifying the overall risk exposure facing your project in this fashion, you are better prepared to justify a realistic contingency buffer that should be built into your schedule.
- This will let you respond to some of the problems that might arise, without totally trashing your timing plans.

Risk Management Can Be Your Friend

- The skillful project manager will use risk management as a technique as a technique for raising the awareness of conditions that could cause her project to go down the tubes.
- Consider a project that begins with an unclear product vision and a lack of customer involvement.
- The astute project manager will spot this situation as posing potential risks, and will document them in the risk management plan.
- By reviewing the risk plan periodically, the project manager can adjust the probability and/or impact of these risks.

- They may rise to the top ten, and can be brought to the attention of senior managers or other stakeholders who are in a position to either stimulate corrective actions, or make a conscious business decision to proceed in spite of the risks.
- We're keeping our eyes open and making informed decisions, even if we can't control every adverse condition facing the project.

Learning from the Past

- While we cannot predict exactly which of the many threats to our projects might come to pass, most of us can do a better job of learning from previous experiences to avoid the same pain and suffering on future projects.
- As you begin to implement risk management strategies on your projects, also keep records of your risk management activities for future reference.

Try these suggestions:

- Record the results of even informal risk assessments, to capture the thinking of the participants on each project.
- Keep records of the mitigation strategies attempted for each of the risks you chose to pursue, noting which approaches worked well and which did not pay off.
- Conduct post-project reviews to identify the unanticipated problems that arose.
- Should you have been able to see them coming through a better risk management approach, or would you likely have been blindsided in any case? Do you think these same problems might occur on other projects? If so, add them to your growing checklist of potential risk factors that the next project can think about.

- Anything you can do to improve your ability to avoid or minimize previous problems on future projects will improve your company's business success and reduce the chaos and frustration that reduces the quality of work life in so many software organizations.

26.9 THE SQA PLAN

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or the software team if a SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project.

A standard for SQA plans has been published by the IEEE [IEE94]. The standard recommends a structure that identifies (1) the purpose and scope of the plan; (2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA; (3) all applicable standards and practices that are applied during the software process; (4) SQA actions and tasks (including reviews and audits) and their placement throughout the software process; (5) the tools and methods that support SQA actions and tasks; (6) software configuration management procedures (Chapter 27) for managing change; (7) methods for assembling, safeguarding, and maintaining all SQA-related records; and (8) organizational roles and responsibilities relative to product quality.

SOFTWARE TOOLS

Software Quality Management

Objective: The objective of SQA tools is to assist a project team in assessing and improving the quality of ... ware work product.

Mechanics: Tools mechanics vary. In general, the intent is to assess the quality of a specific work product. Note: a wide array of software testing tools (see Chapters 13 and 14) are often included within the SQA tools category.

Representative Tools[9]

ARM, developed by NASA (satc.gsfc.nasa.gov/tools/index.html), provides

9 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

26.10 SUMMARY

Software quality management is an umbrella activity—incorporating both quality control and quality assurance—that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

SQA is complicated by the complex nature of software quality—an attribute of computer programs that is defined as "conformance to explicitly and implicitly specified requirements." But when considered more generally, software quality encompasses many different product and process factors and related metrics.

Software reviews are one of the most important quality control activities. Reviews serve as filters throughout all software engineering activities, removing errors while they are relatively inexpensive to find and correct. The formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors.

To properly conduct software quality assurance, data about the software engineering process should be collected, evaluated, and disseminated. Statistical SQA helps to improve the quality of the product and the software process itself. Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

In summary, we recall the words of Dunn and Ullman [DUN82]: "Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering." The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.

CHAPTER 3

SOFTWARE PROJECT MANAGEMENT

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

Software project management is a very vast topic. In fact, a course for a full semester can be conducted on effective techniques for software project management. However, in this chapter, we shall restrict ourselves to only some basic issues.

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

Should the responsibility of software project management rest always on a dedicated full-time project manager? Large projects usually have full-time project managers. However, for small software development projects, one of the software developers assumes the responsibilities of software project management in addition to his normal responsibilities.

In this chapter, we discuss the important responsibilities and activities of a software project manager. We start with a discussion on the scope of the work responsibilities of a project manager. Subsequently, we provide an overview of the planning activity and the organization of the project plan document. We then discuss estimation and scheduling techniques. Finally, we provide an overview of the risk and configuration management issues.

3.1 RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

In this section, we examine the principal job responsibilities of a project manager and the skills necessary to accomplish these.

3.1.1 Job Responsibilities of a Software Project Manager

Software project managers take the overall responsibility of steering a project to success. This surely is a very hazy job description. But, it is very difficult to objectively describe the job

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

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

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

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

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

3.1.2 Skills Necessary for Software Project Management

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

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

3.2 PROJECT PLANNING

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

1. **Estimation:** The following project attributes have to be estimated.
   (i) *Cost* How much is it going to cost to develop the software?
   (ii) *Duration* How long is it going to take to develop the product?
   (iii) *Effort* How much effort would be required to develop the product?

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

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

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

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

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

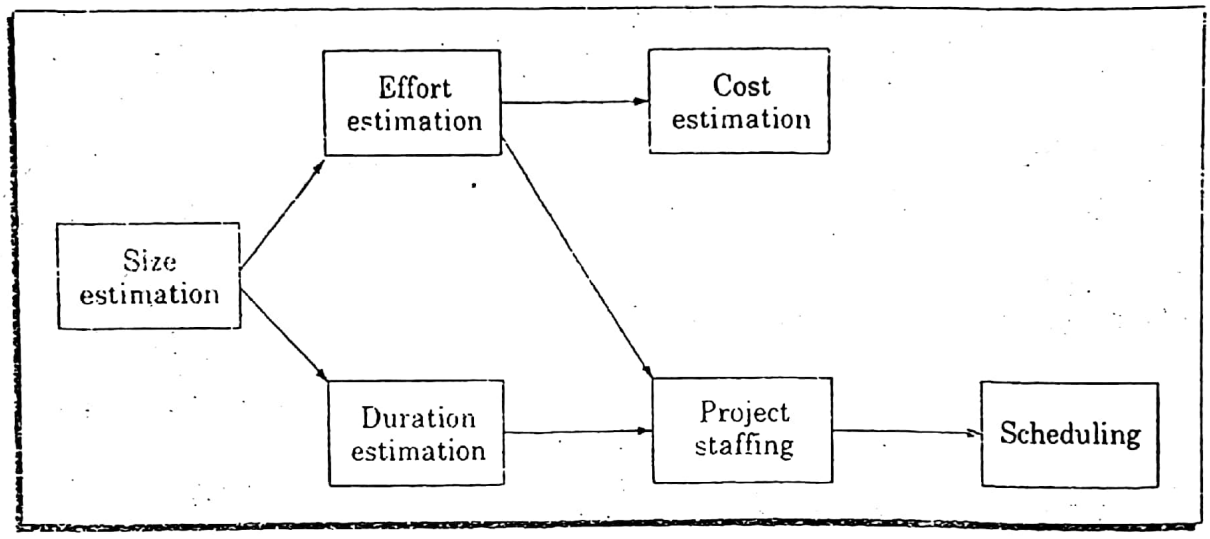

Figure 3.1: Precedence ordering among planning activities.

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

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

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

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissat-

isfaction and adversely affect team morale. It can even cause project failure. For this reason, project planning is considered to be a very important activity. However, for effective project planning, in addition to the knowledge of the various estimation techniques, past experience is crucial.

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

3.2.1 The SPMP Document

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

Organization of the Software Project Management Plan (SPMP) Document:

1. Introduction
   (a) Objectives
   (b) Major Functions
   (c) Performance Issues
   (d) Management and Technical Constraints
2. Project estimates
   (a) Historical Data Used
   (b) Estimation Techniques Used
   (c) Effort, Resource, Cost, and Project Duration Estimates
3. Schedule
   (a) Work Breakdown Structure
   (b) Task Network Representation
   (c) Gantt Chart Representation
   (d) PERT Chart Representation
4. Project resources
   (a) People
   (b) Hardware and Software
   (c) Special Resources

3.5.2 Delphi Cost Estimation

Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The coordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

3.6 COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

COCOMO (COnstructive COst estimation MOdel) was proposed by Boehm, 1981. Boehm postulated that any software development project can be classified into any one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product 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 classes correspond to application, utility and system programs, respectively. Normally, data processing programs[1] 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 typically involve meeting timing constraints and concurrent processing.

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

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

1. **Organic:** We can consider a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development

[1] A data processing program is one which processes large volumes of data using a simple algorithm. An example of a data processing application is a payroll software. A payroll software computes the salaries of the employees and prints cheques for them. In a payroll software, the algorithm for pay computation is fairly simple. The only complexity that arises while developing such a software product arises from the fact that the pay computation has to be done for a large number of employees.

team is reasonably small, and the team members are experienced in developing similar types of projects.

2. **Semidetached:** A development project can be considered 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.

3. **Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if stringent regulations on the operational procedures exist.

Observe that Boehm in addition to considering the characteristics of the product being developed, considers the characteristics of the team members in deciding the category of the development project. 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 KLOC (Kilo Lines of Source Code). One person-month is the effort an individual can typically put in a month. This effort estimate takes into account the productivity losses that may occur due to lost time such as holidays, weekly offs, coffee breaks, etc.

Note that effort estimation is expressed in units of person-months (PM). 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 person-month unit indicates the work done by one person working on the project for one month. It should be carefully noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither nor does it imply that 1 person should be employed for 100 months. The effort estimation simply denotes the area under the person-month curve (see Figure 3.3) for the project. The plot in Figure 3.3 shows that different number of personnel may work at

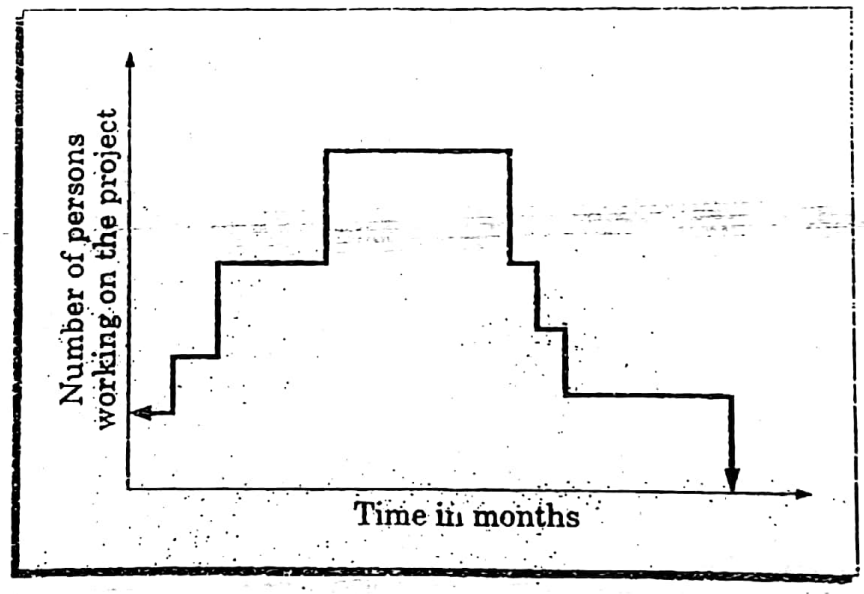

Figure 3.3: Person-month curve.

different point in the project development, as is typical in a practical industry scenario. The number of personnel working on the project usually increases and decreases by an integral

number, resulting in the sharp edges in the plot. We shall elaborate in Section 3.8 how the number of persons to work at any time on the product development is determined.

According to Boehm, software cost estimation should be done through three stages: basic COCOMO, intermediate COCOMO, and complete COCOMO. We discuss these stages as follows:

3.6.1 Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\text{Effort} = a_1 \times (KLOC)^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (Effort)^{b_2} \text{ Months}$$

Where

(a) KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

(b) a_1, a_2, b_1, b_2 are constants for each category of software products,

(c) Tdev is the estimated time to develop the software, expressed in months,

(d) Effort is the total effort required to develop the software product, expressed in person months (PMs).

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a_1, a_2, b_1, b_2 for different categories of products as given by Boehm [1981]. He derived the above expressions by examining historical data collected from a large number of actual projects. The theories given by Boehm were:

Estimation of development effort

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

Organic : Effort $= 2.4(KLOC)^{1.05}$ PM

Semidetached : Effort $= 3.0(KLOC)^{1.12}$ PM

Embedded : Effort $= 3.6(KLOC)^{1.20}$ PM

Estimation of development time

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

Organic : Tdev $= 2.5(Effort)^{0.38}$ Months

Semidetached : Tdev $= 2.5(Effort)^{0.35}$ Months

Embedded : Tdev $= 2.5(Effort)^{0.32}$ Months

We can gain some insight into the basic COCOMO model, if we plot the estimated characteristics for different software sizes. Figure 3.4 shows a plot of estimated effort versus product size. From Figure 3.4, we can observe that the effort is somewhat superlinear (slope of the curve ≥ 1) in the size of the software product. This is because the exponent in the effort expression is more than 1. Thus, the effort required to develop a product increases rapidly with project size. However, observe that the increase in effort with size is not as bad as that was portrayed in Chapter 1. The reason for this is that COCOMO assumes that projects are carefully designed and developed by using software engineering principles.

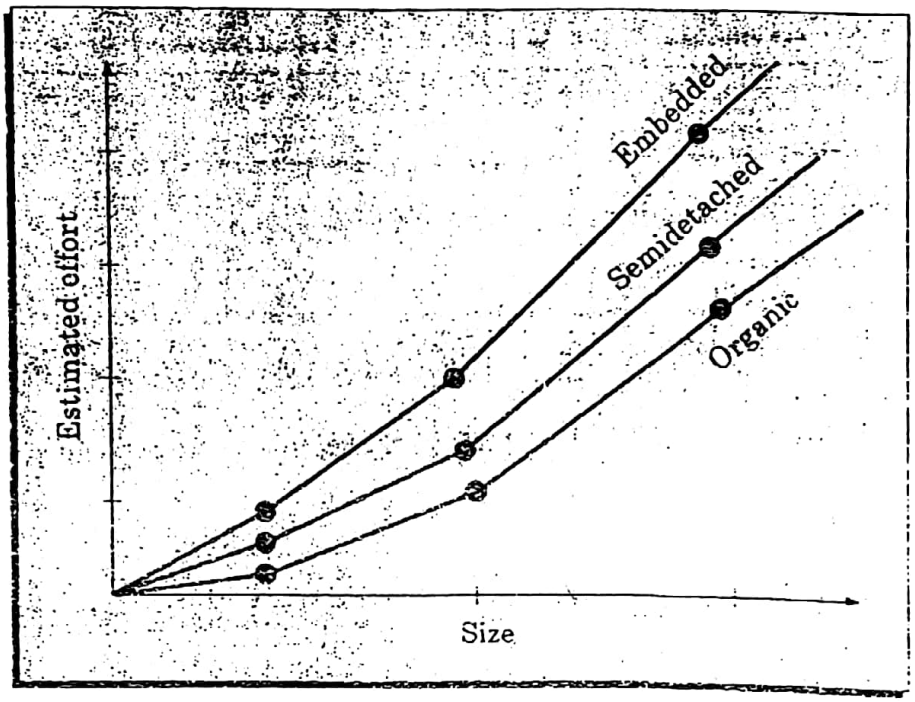

Figure 3.4: Effort versus product size.

The development time versus the product size in KLOC is plotted in Figure 3.5. From Figure 3.5, we can observe that the development time is a sublinear function of the size of the product. That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately. It may appear surprising that the duration curve does not increase superlinearly. The apparent anomaly can be explained by the fact that COCOMO assumes that a project is carried out not by a single person but by a team of developers.

It is important to note that the effort and duration estimations obtained using the COCOMO model imply that if you try to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if you complete the project over a longer period of time than that estimated, then there is almost no decrease in the estimated cost value. The reasons for this are discussed in Section 3.8. Thus, we can consider that the computed effort and duration values to indicate the following.

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

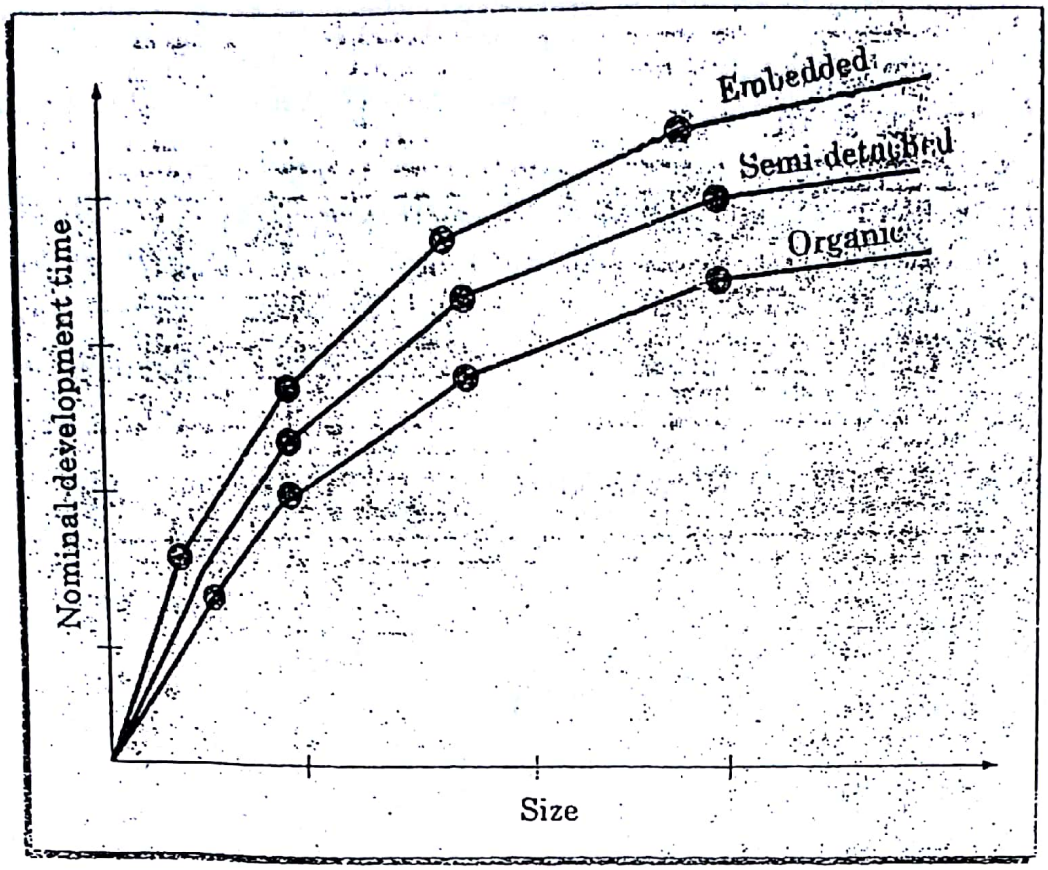

Figure 3.5: Development time versus size.

An optimum sized team is one in which developers do not have to wait idle waiting for work but at the same time employs as many developers as possible. That is why the duration given by COCOMO is called the nominal duration or the optimal duration. It is called optimal duration, if the project is attempted to be completed in a shorter time, then the effort required would rise rapidly. This may appear as a paradox, since the same product would be developed though over a shorter time, then why should the effort required rise rapidly? This can be explained by the fact that for every product, there is a limit on the number of parallel activities that can be identified and carried out. Thus, if more number of developers are deployed than the optimal size, some of the developers would have to idle for some time since it would not be possible to give them any work. These idle times would show up as higher effort and larger cost.

From Figure 3.5, we can observe that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type. (Please verify this using the basic COCOMO formulas provided above) We can interpret it to mean that there is more scope for parallel activities for system and utility programs than those in application programs.

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

Another point should be carefully remembered. From the effort estimation for a project expressed in programmer-months and the nominal development time, the novices would de-

termine the staffing level by a simple division. However, we are going to examine the staffing problem in more detail in Section 3.8. From the discussion in Section 3.8 it would become clear that the simple division approach to obtain the staff size is highly improper.

Example 3.1 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 software developers 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 = $2.4 \times (32)^{1.05} = 91$ PM

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

Cost required to develop the product = $91 \times 15,000$ = Rs. 1,465,000

3.6.2 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 required to develop the product as well as the development time. 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 recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. 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, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

1. **Product:** The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.
2. **Computer:** Characteristics of the computer that are considered include the execution speed required, storage space required, etc.
3. **Personnel:** The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.
4. **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.

We have discussed only the basic ideas behind the COCOMO model. A detailed discussion on the COCOMO model are beyond the scope of this book and the interested reader may refer [Boehm, 81].

3.6.3 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 subsystems. These subsystems may have widely different characteristics. For example, some subsystems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also 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 subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

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 organization having offices at several places across the country can have the following subcomponents:

- 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 semidetached 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 improve the accuracy of their results, the different parameter values of the model can be fine-tuned and validated against an organization's historical project database to obtain more accurate estimations. Estimation models such as COCOMO are not 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. Companies consider computed estimates to be satisfactory, if these are within about 80% of final cost. Although these estimates are gross approximations—without such models, one has only subjective judgements to rely on.

3.6.4 COCOMO 2

Since the time that COCOMO estimation model proposed in the early 1980s, both the software development paradigm and problems have undergone a sea change. The present day software projects are much larger in size and reuse of existing software to develop new products has become pervasive. This has given rise to component-based development. New life cycle models and development paradigms are being deployed for web-based and component-based software. During the 1980s rarely any program was interactive, and graphical user interfaces were almost non-existent. On the other hand, most of the present day software is highly interactive and has elaborate graphical user interface. To make COCOMO suitable in the changed scenario, Boehm proposed COCOMO 2 [Boehm, 95].

COCOMO 2 provides three increasingly detailed cost estimation models. These can be used to estimate project costs at different phases of the software. As the project progresses through these models can be applied at the different stages of the same project.

1. **Application composition:** This model as the name suggests, can be used to estimate cost for prototyping, e.g. to resolve user interface issues.
2. **Early design:** This supports estimation of cost at the architectural design stage.
3. **Post-architecture stage:** This provides cost estimation during detailed design and coding stage.

The post-architectural model can be considered as an update of the original COCOMO. We discuss these three models in the following.

Application composition model

The application composition model is based on counting the number of screens, reports and 3GL modules. Each of these components is considered to be an object (this has nothing to do with the concept of objects in the object-oriented paradigm). These are used to compute the object points of the application.

Effort is estimated in the application composition model as follows:

1. Estimate the number of screens, reports and 3GL components from an analysis of the SRS document.
2. Determine the complexity level of each screen and report, and rate these as either simple, medium, or difficult. The complexity of a screen or a report is determined by the number of tables and views it contains.

Table 3.2: SCREEN complexity assignments for the number of views and data tables

Number of views	*Tables* < 4	*Tables* < 8	*Tables* ≥ 8
< 3	simple	simple	medium
3–7	simple	medium	difficult
>8	medium	difficult	difficult

3. Use the weight values in Tables 3.2 to 3.4.

Table 3.3: Report complexity assignments for the number and source of the data tables

Number of sections	*Tables* < 4	*Tables* < 8	*Tables* ≥ 8
0 or 1	simple	simple	medium
2 or 3	simple	medium	difficult
4 or more	medium	difficult	difficult

The weights are supposed to correspond to the amount of effort required to implement an instance of an object at the assigned complexity class.

4. Determine the number of object points
   Add all the assigned complexity values for the object instances together—The Object Count.
5. Estimate percentage of reuse expected in the system (reuse refers to the amount of pre-developed software that will be used within the system). Then, evaluate New Object-Point count (NOP),

the Putnam's model indicates an extreme penalty for schedule compression and an extreme reward for expanding the schedule. However, increasing the development time beyond some optimal time has been found to increase the total effort rather than decrease it.

The Putnam estimation model works reasonably well for very large systems, but seriously overestimates the effort on medium and small systems. This is also corroborated by Boehm [Boehm, 81]. Boehm states that there is a limit beyond which a software project cannot reduce its schedule by buying any more personnel or equipment. This limit occurs roughly at 75% of the nominal time estimate for small and medium sized projects. Thus, if a project manager accepts a customer demand to compress the development schedule by more than 25% he is very unlikely to succeed. The main reason being that every project has only a limited amount of activities which can be carried out in parallel and the sequential activities cannot be speeded up by hiring any number of additional developers.

3.8.4 Jensen's Model

Jensen model [Jensen, 84] is very similar to Putnam model. However, it attempts to soften the effect of schedule compression on effort to make it applicable to smaller and medium sized projects. Jensen proposed the equation:

$$L = C_{te} t_d K^{1/2}$$

or

$$K_1/K_2 = t_{d_2}^2/t_{d_1}^2$$

where, C_{te} is the effective technology constant, t_d is the time to develop the software, and K is the effort needed to develop the software.

When the project duration is compressed, Jensen models the increase in effort (and cost) requirement to be proportional to the square of the degree of compression.

3.9 SCHEDULING

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

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

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

3.12 RISK MANAGEMENT

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

3.12.1 Risk Identification

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

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

Project risks

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

Technical risks

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

Business risks

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

Example classification of risks in a project

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

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

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

3.12.2 Risk Assessment

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

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

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

$$p = r * s$$

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

3.12.3 Risk Containment

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

Avoid the risk

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

Transfer the risk

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

Risk reduction

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

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

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

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

engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to set a milestone every 10 to 15 days.

3.13 SOFTWARE CONFIGURATION MANAGEMENT

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

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

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

3.13.1 Necessity of Software Configuration Management

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

Inconsistency problem when the objects are replicated

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

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

Problems associated with concurrent access

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

Providing a stable development environment

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

System accounting and maintaining status information

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

Handling variants

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

3.13.2 Configuration Management Activities

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

Configuration management is carried out through two principal activities:

- **Configuration identification** involves deciding which parts of the system should be kept track of.
- **Configuration control** ensures that changes to a system happen smoothly.