Project Management measures for Small Enterprises

Master Thesis, 2005

by

Mushtaq Ahmed Mohammed

Department of Computing Science
Umeå University
SE-90187 Umeå, Sweden

Submitted to the Department of Computing Science at Umeå University in partial fulfillment of the requirements for the degree of Master of Science in Computing Science.

Thesis Defense: April 1st 2005, MIT building.
Thesis Supervisor: Annabella Loconsole, Department of Computer Science, Umeå University, Sweden.

(This page is left intentionally)

Abstract

Project management faces many difficulties in developing a project when suitable measures are not applied. Different methods and measurement tools like COCOMO, SLIM and analogy methods etc. are available in the literature for managing enterprises. However current research is more focused on large-scale enterprises while there are no specific tools and methods for small-scale enterprises. Therefore it is essential to do more research specific to the project management and measurement in small enterprises.

My Contribution to this research is to describe the concepts of software project management, software project measurements and their relation, to classify the organizations regarding their size based on number of employees and number of products under development. I have compared the measurements used in large scale organizations with properties of small organizations and new organizations and suggested suitable measures for the small scale enterprises. For the above comparison tools like COCOMO, SLIM have been used. As a result of the comparison the project management measures vary with the size of the organization

Acknowledgements

I would like to acknowledge my supervisor Annabella Loconsole for fully supporting me during my thesis, giving me valuable advices, being open minded to discuss various concepts, teaching me the approach to do research and being a friendly and adjustable. I once again thank you for the cooperation made by you.

I would like to thank Mr. Per Lindström for supporting and encouraging me from the day I enter the university, his helpful nature gave me good support to with stand environmental conditions and complete my thesis work.

I would like to thank Mr. Jürgen Böstler who has taught me objected oriented software development course for his support to complete the course.

I thank and express my gratitude towards my Uncle Mr. Mohammed Jabbar Ahmed who has been a great support in every aspect trough out the degree, without his support I could not complete this degree. Finally I thank my parents, guardians and friends who have been my strength and moral support for the completion of thesis.

List of Figures

Abbreviations

1. COCOMO: *Constructive COst MOdel*
2. CLOC: *Commented Lines of Code*
3. CMM: *Capability Maturity Model*
4. ELOC: *Effective Lines of Code*
5. EAF: *Effort Adjustment Factor*
6. FP: *Function Point*
7. KSLOC: *Kilo Source Lines of Code*
8. LSO: *Large Scale Organization*
9. QMSE: *Quality Management for Small*
10. S: *Small*
11. SE: *Small Enterprises*
12. SLOC: *Source Lines of Code*
13. SLIM: *Software Life Cycle Measure*
14. TFC: *Total Complexity Factor*
15. UFC: *Unadjusted Function Point Count*
16. XXS: *eXtra eXtra Small*
17. XS: *eXtra Small*

‘

Introduction

1.1 Background

Computers have become a part of everyday life. They are used in homes, universities, banks, hospitals, offices etc. Computers are slowly but surely becoming a part of most of the buildings in this modern era. It is impossible to imagine a building without electricity or electric lamps. They are very important components of a building and are invisible and transparent in use. Computers are along the same path as well, but are in a period of transition from being in buildings to being a part of the buildings. In this journey of new era, software development plays an important role because we use computers in our daily life to do our work with ease and to save our time. To use computers means to use different softwares to perform different functions for lessening our burden. And therefore software development has major part to do. As we enter the software development the first and foremost thing which plays an important role for the success of the project or the software developed is project management, without the proper project management it is almost impossible to develop the software within budget and time. And the project management without proper measurement cannot complete the project within the time and budget. This is the case in both large-scale organizations and small-scale organizations. However lot of research is done for large-scale organization while the interest for the SE's is low. Therefore I focused my research on small-scale enterprises.

1.2 Motivation

Now a days use of software have become essential in our daily life and offices. The organizations which develop software must produce good quality product to satisfy the customer and moreover within the budget, to be in profit. If there is no profit it is mere waste to run an organization. The research has been always focused for the success of large-scale organizations (LSO). Even though the researches have been focused in the LSO until now we have no specific methods or measures to measure all the attributes in the project. Hence it is necessary to do research in the small-scale enterprises to define some suitable measures for the project management to make successful software. The success of project is directly proportional to the project management and the measures taken by it.

1.3 Concepts Used

Software project Management: The main aim of software project management is to deliver high quality project within the budget and time. And the tasks that project management handles are planning and scheduling project activities, estimating resource requirements, staffing, co-ordinate activities and resources, dealing with deviations from the plan and choosing suitable metrics according to the situation. It is hard to manage the project because of the

following reasons creative and human intensive activity, adjusting to requirements of perfection, preservation of conceptual integrity of the product with the division of labour, development of software product costs several times as much as debugged program with same function, manage conflicting goals by division of work to reduce delay in delivery vs. delay due to integration of work and communication, techniques of resource estimation are poorly developed and due to week estimates the day by day slippages in the schedule delays the project by years [Pressman, 1997][Tom Gilb, 1998] [Wood ward S, 1999].

Software project measurement: This area deals with metrics used by the project management to make the project successful. Software measurement is a process, which provides information in time that improves decision making to affect the business or mission outcome. Software measurement is about the assessment and prediction of well-defined *attributes* of well-defined *entities* (i.e., process, product, resource). If the project management needs to produce good quality software product their software measurements must be good. How good is a program? How reliable will a software system be once it is installed? How much more testing should we do? How many more bugs can we expect to find? How much wills the testing cost? How difficult will it be to maintain a system? How much will it cost to build a new system similar to one we built five years ago? How long will it take? These are the questions, which give rise to software measurement. To answer these questions project management suggest suitable measures and perform the measurement, according to the result obtained it takes the decision to make the project successful. And I have used tools like COCOMO and SLIM which are used in large scale organisations and analysed whether they are suitable for estimations in small scale enterprises by considering their properties [Norman Fenton, 1995][Jones C, 1996][[Tom Gilb, 1998].

1.4 Contribution

As the goal of my thesis is to evaluate project management measures for the Small scale enterprises, my contributions to this thesis are

1. A description of project management with respective to different issues.
2. A discussion of project management and its relation with project measurement.
3. A description of importance of project measurement and use of metric information by managers, engineers and customers.
4. COCOMO and SLIM models are described.
5. A classification of small organisations into XXS, XS and S.
6. Project management measures for XXS, XS and S enterprises are presented.

1.5 Thesis Outline

Chapter2- Discusses the project management and its relation with measurement and its importance in present software development in current conditions.

Chapter3- Discuss the project measurement and its importance in current conditions, Explains COCOMO and SLIM models. Project management with measurement.

Chaper4- Describes the software cost estimation process and explains COCOMO and SLIM models.

Chapter5- Evaluation of different methods and tools for SE considering measurable factors in software development, classification of small scale organization.

Chapter6- Conclusion is presented

Project management

2.1 Introduction

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development, and maintenance of computer software. In order to organize and manage a software development project successfully, one must combine specific knowledge, skills, efforts, experience, capabilities, and even intuition. Software Project Management differs from other project managements in several ways. The Project Manager, like a captain on a ship, is ultimately responsible for the success of a project. The project manager, besides being the focal point for the project execution, is responsible for driving organizational initiatives and changes, and keeping the customer delighted all the time. Most of the software projects fail not because they are inherently complex, but mainly because of the poor project management. You can be a smart programmer capable of writing complex programs, but it is not enough for project management. The project management skills are quite different. It involves getting the work done by others, the way you want and yet making them feel happy about it [Ahituv, N., Zviran, M., Glezer, C, 1999].

Software project management focuses on the four

1) People
2) Problem
3) Process
4) Time

People: Project management organises the people into effective teams, motivate them to do high-quality software work, and coordinate them to achieve effective communication. The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development [Ahituv et al., 1999].

Problem: The Project Management communicates the problem from customer to developer partitioned into its constituent parts. It establishes objectives and scope of the project before it is planned, it considers alternative solutions and identifies technical and management constraints. Finally it gives the manageable project schedule that gives meaningful indication of the process [Ahituv et al., 1999].

Process: Project management focuses on software process because it provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all

software projects, regardless of their size or complexity. The process must be adapted to the people and the problem [Ahituv et al., 1999].

Time: It makes schedule plans in order to complete the project at the required time. If it doesn't focus on time it increases schedule which results in delay of deliverable and hence will not be able to produce the product in time. [Ahituv et al., 1999]

The main aim of software project management is to deliver high quality project with in the budget and time. Software development is a complex process involving activities as domain analysis, requirements specification, communication with the customers and end-users, designing and producing different artifacts, adopting new paradigms and technologies, evaluating and testing software products, installing and maintaining the application at the end-user's site, providing customer support, organizing end-user's training, envisioning potential upgrades and negotiating about them with the customers, and many more.

In order to keep everything under control, eliminate delays, always stay with in the budget, and prevent project runaways, i.e. situations in which cost and time exceed what was planned, software project managers must exercise control and guidance over the development team throughout the project's lifecycle [Ahituv et al., 1999]. In doing so, they apply a number of tools of both economic and managerial nature. The first category of tools includes budgeting, periodic budget monitoring, user charge back mechanism, continuous cost/benefit analysis, and budget deviation analysis. The managerial toolbox includes both long-range and short-term planning, schedule monitoring, feasibility analysis, software quality assurance, organizing project steering committees, and the like. All of these activities and tools help manage a number of important issues in the process of software development. Figure 1 illustrates some of the issues, but definitely not all of them. The issues shown in Figure 1 have been selected for an extended overview in the remainder of this chapter based on the following criteria:

1. Their priority in the concerns of most software project managers, according to the managers themselves - this is evident from the case studies, interviews, and reports of many software project managers and consultants in software industry worldwide (see, for example, [Booch, 1996], [Fayad et al., 1996], and [Woodward, 1999]).

2. The frequency of their appearing as topics in the relevant, industry oriented software engineering journals and magazines, such as *IEEE Computer*, *IEEE Software*, and *Communications of the ACM*, during the last decade.

3. Their importance as identified by relevant committees, associations, and consortia of software developers (see for example, [SWEBOK, 2000]).

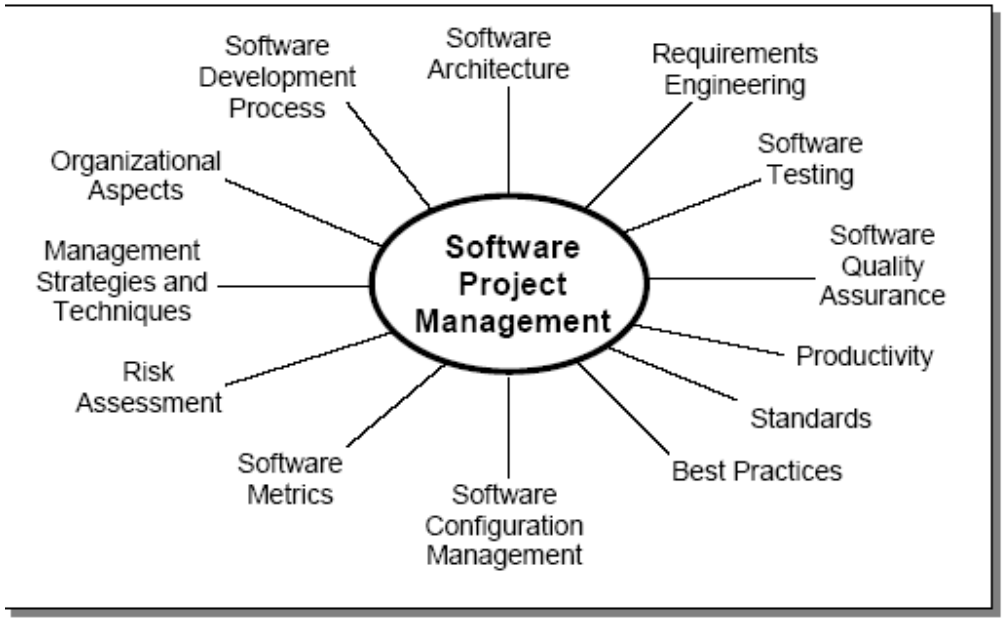

Figure 2.1: Project Management issues [Booch 1995] [Fayad et al...1996][Woodward 1999].

These are the some of the aspect on which Project management focuses but as my thesis is more concerned on measurement side of the project management I first describe why the project management chooses the software metrics and how they are related then the rest of the aspects.

2.2 Project management with measurement

Measurement is a key factor for managing and improving software development. The purpose of the measurement process in software projects is to define and operate a context-specific set of metrics, and to describe the required guidelines and procedures for data collection and analysis [Rettig M., Simons G, 1993].

One question that most managers have is how much a process cost ? If we can measure the time and the effort we use in the software production, we can also understand not only the cost of the total project, but also how the different parts contribute to the whole. When we have a measure of cost or effort we can also combine this with a measure of size, in order to get a notion of the productivity of the project. By collecting productivity information about a large set of projects, We can then make attempts to predict the cost and duration of future projects.
Software metrics' information can also be used to evaluate the quality of the products we use. For instance, different kinds of fault measures can be used to assess the code being produced for comparing, predicting and setting targets for process and product improvement. But the relative occurrence of faults is not the only measure of software quality. In order to satisfy the customer we also need a high level of functionality. If we can measure usability, reliability, response time,

and other characteristics we can also find out if our customers will be happy with both functionality and performance [Fenton & Pfleeger, 1997].

An overall objective of most managers is to improve their business by focusing on those activities, which will generate the largest revenue relative to the cost of the improvement. If we can measure the time it takes to perform each development activity, and calculate its effect on quality and productivity, we can also weigh the costs and benefits of each practice to determine if the benefit is worth the cost. Another way of improving the business is for example to compare two different design methods and measure the results to decide which one is the best. Software measurement generates quantitative descriptions of key processes and products, enabling project management to understand behaviour and result. Such descriptions can indicate the effort needed to complete the project, the product's quality, estimated schedules and time-to-market, rework effort, estimated project costs, and distribution of resources and costs by project phases. Software measurement makes possible to compare the project on which a development team is currently working on, to similar projects in terms of budget, costs, productivity, quality, staffing, development processes, and technology used [Pfleeger et al., 1997].

In order to operate a metrics program during a software development project, the project manager must enforce continuous measurement of relevant factors. These factors depend on the overall management goals of the measurement process. In that sense, one can differentiate between the following kinds of software metrics [Albrecht, A.J., Gaffney, 1983], [Boehm, B.W., et al, 1985], [Chidamber, S.R., Kemerer, C.F, 1994], [Deveaux, P, 1993], [Jones, C, 1991],[Kulik, P, 2000], [Lavazza, L, 2000], [Whitmire, S.A, 1993].

1. ***Metrics for project size and team productivity*** – typical and most widely used representatives of this kind of metrics are *source lines of code (SLOC)* and *function points*; the SLOC metric can be converted relatively accurately and easily into the number of programmer-months needed to complete the project; function points are dimensionless numbers that indicate the application's functionality from the user's perspective, and can also be easily converted into the effort needed to complete the project or one of its parts.

2. ***Metrics for schedules*** – these include the number of tasks not completed on time, the number of tasks with changed schedules, and the number of postponed tasks.

3. ***Metrics for requirements specification*** – the number of requests for change (RFC) in specification, the number of new requirements, and the RFC diagram (showing the dynamics of RFC over time).

4. ***Metrics for software testing*** – these metrics are used to track the percentage of SLOC covered by the testing process; increasing that percentage reduces the number of errors to be discovered by the users and increases the product's quality.

5. ***Metrics for software quality*** – they typically show the fault density (the number of errors per 1 KSLOC) and fault arrival and closing rates; as a rule of thumb, the product's quality is satisfactory if the fault density is lower than 0.25.

6. ***Metrics for project risk*** – they measure confidence in the product's ready to- deployment date (over time taken).

The most widely used metrics models include COCOMO [Boehm, B.W., et al], which is based on measuring SLOC, function points analysis [Albrecht, A.J., Gaffney], [Deveaux, P.], [Jones, C, 1991], [Whitmire, S.A, 1993], GQM (Goal-Question-Metrics, based on systematic translation of the company's goals into the measurement process goals, and refinement by defining the concrete measurements to perform in order to support the goals) [Lavazza, L. June 2000], and Chidamber-Kemerer's metrics suit for object-oriented software projects (specifying metrics for the number of methods per class, depth of inheritance tree, number of children, etc.) [Chidamber, S.R., Kemerer, 1994].

These are measures used by project management at different stages to have a vivid picture of the situation for making decision.

2.3 Project Management Issues

2.3.1 SW Development Process

One of the primary duties of the manager of a software development project is to ensure that all of the project activities follow a certain predefined *process*, i.e. that the activities are organized as a series of actions conducing to a desirable end. The activities are usually organized in distinct *phases*, and the process specifies what artefacts should be developed and delivered in each phase. A software development team conforming to a certain process means complying with an appropriate *order* of actions or operations. For the project manager, the process provides means for control and guidance of the individual team members and the team as a whole, as it offers criteria for tracing and evaluation of the project's deliverables and activities. Software development process encompasses many different tasks, such as domain analysis and development planning, requirements specification, software design, implementation and testing, as well as software maintenance. Hence it is no surprise at all that a number of software development processes exist. Generally, processes vary with the project's goals (such as time to market, minimum cost, higher quality and customer satisfaction) and available resources (e.g., the company's size, knowledge, and experience of people -both engineers and support personnel -- and hardware resources).

However, every software developer and manager should note that processes are *very* important. It is absolutely necessary to follow a certain predefined process in software development. It helps developers understand, evaluate, control, learn, communicate, improve, predict, and certify their work. Since processes vary with the project's size, goals, and resources, as well as the level at which they are applied (e.g., the organization level, the team level, or the individual level), it is always important to define, measure, analyse, assess, compare, document, and change different processes [Lindvall, M., Rus, 2000].

There are several well-known examples of software development processes like waterfall model, Spiral model etc.

2.3.2 Requirements Engineering

Requirements engineering is the discipline of gathering, analysing, and formally specifying the user's needs, in order to use them as analysis components when developing a software system [Davis, A.M., Hsia, P, March 1994], [Siddiqi, J,1994]. Requirements *must* be oriented towards the user's real needs, not towards the development team and the project managers.

Almost all software development processes one way or another stress requirements analysis and specification as one of their core workflows. The reasons are simple. It is necessary to manage requirements possible because a small change to requirements can profoundly affect the project's cost and schedule, since their definition underlies all design and implementation [Reifer, D.J, June 2000]. Unfortunately, in most practical projects it is not possible to freeze the requirements at the beginning of the project and not to change them. Requirements develop over time, and their development is a learning process, rather than a gathering one. The intended result of this process is a structured but evolving set of agreed, well understood, and carefully documented requirements [Jarke, M,1998]. This implies the need for requirements *traceability*, i.e. the ability to describe and follow the life of a requirement, in both a forward and backward direction, ideally through the whole system's life cycle.

2.3.3 Software Architecture

Software architecture encompasses specification and design of the application's global structure, leaving the details aside [Shaw, M., Garlan, 1996]. It is related to the general software organization in terms of its *components* and *connectors*. Components are things like modules, compilation units, objects, and files. Connectors define interactions among components through procedure calls, parameters of initialisation, instructions for the linker, and so on.

Defining the architecture of a software system involves the choice of *architectural style*. Each architectural style defines a family of software systems organized in a similar way, the corresponding vocabulary of components and

connectors, constraints in using the components and connectors in building the system according to that style, and the way the overall system behaviour depends on the behaviour of its components. Examples of software architectural styles include layered architectures, pipeline architecture, object-oriented architecture etc.

As soon as the initial set of requirements is gathered, the project manager should direct the chief architect and some other engineers to define the initial software architecture of the system to be developed. Software architecture definition does not get sealed after the project begins. On the contrary, it is an evolving activity that continues through all the phases of the product's lifecycle. It interweaves with requirements specification, domain analysis, study of possibilities for reuse, and even design.

Selecting an architectural style and evolving the software architecture is far from being simple, because it involves many issues other than just the system's overall structure. Project managers must be aware of such issues. Some of the issues include the platform the system is to run on (e.g., the hardware architecture, operating system, database management system, and network protocols), global control structures, data communication, synchronization, and access protocols, etc. For that reason, the architecture of a software system is considered a product in its own right, along with the main software product to be delivered to the customers. It is the development team that benefits most from the software architecture as a product [Jacobson, I., Booch, G., Rumbaugh, 1999].

2.3.4 Organizational Aspects

Software project management always involves various organizational aspects, such as creating and staffing development teams, assigning roles to the team members, modalities of software development, leadership considerations, interpersonal communication at work, staff training and embracing new technologies, organization's culture, social and ethical issues, and so on. Organizational aspects of software development are crucial for all successful projects. They are neither about hardware nor about software – they are about "people ware, i.e. using, coordinating, and managing human resources in an organization effectively. In the context of fast-paced and extremely fluid dynamics of software industry, the key to success or failure of software project is the way it is organized and managed [Constantine, L.L, 1993] [Athey, T, 1998], [Booch G 1996].

2.3.5 Management Strategies and Techniques

Software development is an extremely dynamic and fluid business, and it is difficult to plan everything at the beginning of a project. Therefore, efficient management of software projects must be based on some explicit, strategic goals and organization's interests. There are a several of useful guidelines to follow. [Athey, T, 1998], [Holtzblatt, K., Beyer, 1993]. Some of them are listed below

1. Balancing the need for structure and process control in software development with the need for flexibility, informality, and more effective communication processes;

2. Establishing software measurement programs and enforcing accountability for completion of software development milestones;

3. Making management objectives and product vision clear to the development team members (this is very important in practice, because far too often developers are in total ignorance of the broader strategy of a company, and the tactical decisions made by management to advance this strategy seem to them arbitrary and even hostile);

4. Identifying the most critical issues of the project and stressing the need to allocate most development resources, time, and efforts to such issues;

5. Organizing more visible and formal management processes for reviewing and approving potential product enhancements;

6. Emphasizing management approaches that facilitate flexibility and creativity within clearly defined boundaries;

7. Keeping-up with technological developments by enforcing life-long learning, training, courses, and seminars;

8. Developing more globally focused, culturally sensitive management capabilities;

9. Involving end-users in the development process in order to constantly provide advice on using the product in the real world, thus eliminating the customer-developer gap;

10. Maintaining *progress charts* that show the percentage of completion for each module, at any given moment of product development;

11. Keeping track of all relevant facts about the product (e.g., previous versions, delivery dates, current version), the development process (the problems encountered, resulting delays, and the reasons why they have occurred), and discarded design alternatives in the *external group memory* (it is usually a special-purpose project-management software, or a site on the organization's server or Intranet, and sometimes even a site on the Internet); the external group memory can also serve as a board for discussion on all the relevant ideas that arise in the course of the project;

12. Estimating time and effort needed for each designer to complete a short-term task, e.g. an iteration in an iterative development process; for that

purpose, each designer may be required to initially fill-up and constantly update a *planning sheet* that contains both the designer's original estimates and actual measures (in days) of how long it takes to complete each activity for the task (activities may include analysis, design, coding, testing)

13. Emphasizing progress review mechanisms across the development effort;

14. Insisting on creating multiple design views, such as structural, functional, object-oriented, event-based, and data-flow; although sometimes redundant, multiple design views help cover design from multiple perspectives and make it more complete and more efficient;

15. Enforcing the feedback mechanism in the development process, in order to detect inconsistencies in design as early as possible and reduce the costs of fixing them.

2.3.6 Productivity

Generally, productivity is an output divided by the effort required to produce that output. In software development, the output is a completed software development project. In order to consider a software company's productivity, it is necessary to somehow translate that output into a meaningful measurement. Ideally, software project managers should base output measurement on a combination of a project's size, functionality, and quality [Maxwell, K.D., Forselius, 2000]. However, such a measurement doesn't yet exist.

2.3.7 Software Testing

Inspite of the fact that in every software development project the product undergoes testing, delivered software always contains residual defects. Software testing is a difficult, time-consuming process. It requires specific skills from software testers, skills that only partially overlap with those of software developers. Apart from mastering coding, testers must also possess a great deal of knowledge of formal languages, graph theory and algorithms [SWEBOK, 2000], [Whittaker, J.A, Feb 2000].

Typically, software testing proceeds in four phases [Whittaker, J.A, Feb 2000].
• modelling the software's environment
• selecting test scenarios
• running and evaluating test scenarios
• measuring testing progress

In the first phase, the tester's task is to simulate the interaction between the application and its environment, be it the user or the other applications, taking into account all possible inputs and outputs that can cross the application's

boundaries. The hardest part here is the fact that in many cases the interactions can go through numerous different file formats, communication protocols, GUIs, and file systems. The other hard part is the unpredictability of the user's actions– the software under test must account for that.

Since the number of possible test scenarios is usually extremely large, testers should select those scenarios that cover all code statements and all significant representatives of external events. Before running the selected scenarios, it is necessary to convert them into executable form (often as code) in order to simulate typical interactions between the system and the external world. Applying test scenarios manually is labor-intensive and error-prone. For that reason, testers try to automate the test scenarios as much as possible. In many environments, automated application of inputs through code that simulates users is possible, and tools are available to help [Mills et al., 1987].

Measuring testing progress is difficult, simply because it is not just counting the numbers of bugs found. As stated in the section on software metrics, specific metrics for software testing are used to measure the *coverage* of the tests applied (in terms of running all lines of the source code, forcing all the internal data to be initialized and used, applying all test scenarios, exploring all the inputs, and checking for functional completeness). Note also that software reliability engineering can greatly help - the *Cleanroom methodology* developed at IBM [Mills et al., 1987] has been particularly useful in improving software quality and providing a quantitative measure for the quality of a software product at its release. The Cleanroom approach provides for the transition of process technology to the project staff and integrates several proven software engineering practices into one methodology [Wayne Sherer et al., 1996]. The testing strategy of the Cleanroom methodology can be best described as random sample based on usage model that predicts field reliability, rather than a futile attempt for coverage and little insight on field reliability.

If the Unified Process is used to manage software development, software testing is performed in every iteration [Jacobson et al., 1999]. Test scenarios are defined from use cases, and comprise both functionality and performance testing. The advantage of this incremental and iterative approach to software testing is that in each iteration the testers test just some of the application. Moreover, the tests performed in early phases usually discover such bugs and faults that would cause more severe instability in the project's rhythm if they were discovered in later phases. In every iteration, tests also check whether the current iteration has jeopardized some of the previously built and tested architecture. If the project's size is large, it is impractical to manually run all the test cases, so the use of automated testing tools is recommended. Project managers should adopt the practice of enforcing thorough testing in every iteration, and not allowing the next iteration to begin before all the tests planned in the current iteration are completed. The entire project is considered completed only when all the UML models *and* all the tests are completed and delivered [Mills et al., 1987] [Jacobson et al., 1999] [Wayne Sherer et al., 1996].

2.3.8 Software Quality Assurance

The goals of software quality assurance (SQA) are monitoring the software and its development process, ensuring compliance with standards and procedures, and ensuring that product, process, and standards defects are visible to management [SWEBOK, 2000].

It is desirable for a software development organization to plan and control product quality *during* development. Projects managers cannot allow the luxury of going back and adding quality - by the time a quality problem is detected, it is probably too late to fix it [Reel, J.S, Hune 1999]. For that reason, it is necessary to establish procedures and expectations for high levels of quality before any other development begins. Also, hiring developers proven to develop high-quality code, staffing the project accordingly, and enforcing peer-level code reviews and external reviews must be top priority of every software project management. Planning and controlling software product quality during development requires [Boegh, J., Depanfilis, S., Kitchenham, B., Pasquini, A, 1999]:

1. Establishing targets for the external quality characteristics;

2. Pursuing those targets during development by defining and monitoring targets for internal quality characteristics - this can be done using conventional software measures of size, fault rates, change rates, structure, test coverage, and so on, taken early in product development;

3. Establishing relationships between internal and external quality characteristics, using experience from similar past software development projects;

4. Identifying and setting targets for internal quality characteristics.

2.3.9 Software Configuration Management

The configuration of a software system is a collection of specific versions of hardware, firmware, or software items combined according to specific build procedures to accomplish a particular purpose [SWEBOK, 2000], [Buckley, F.J, 1996].

Software configuration management (SCM) comprises a set of technical, managerial, and administrative activities related to identifying the configuration of a software system at distinct points in time for the purpose of systematically controlling changes to the configuration, recording and reporting change processing and implementation status, verifying compliance with specified requirements, and maintaining the integrity and traceability of the configuration throughout the system life cycle [Bersoff, E.H, 1997]. Responsibilities of each software project manager related to SCM include enforcing the practice of SCM

activities for the project, distributing the activities to the relevant individuals, and managing and administrating the results of these activities.

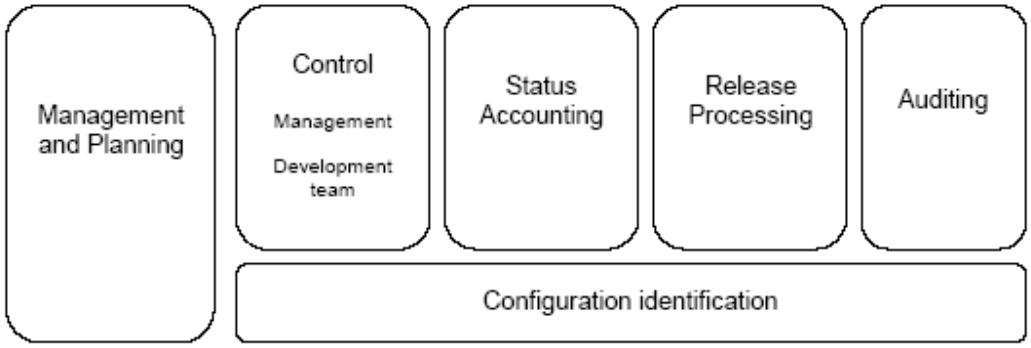

Figure 2.2: Software configuration management activities [SWEBOK, 2000]

2.3.10 Risk Assessment

In order to prevent project runaways, meet deadlines, stay within the project's budget, and simultaneously maintain the product's high quality standards, it is essential to timely identify and periodically evaluate certain critical factors. Such factors include [Ahituv et al., 1999], [Boehm, 1991], [Jones, 1996], [Keil et al., 1998]:

1. Estimating the project's size in the early phases - the project's size affects how the deadlines will be set up, and is positively correlated with monetary expense and risk.

2. Setting up the deadlines realistically - as a result, the necessary time to establish the rhythm of the project, prevent delays, and enter a steady state in which the effort is equally distributed from the beginning of the project, without putting an extra workload to the team members at the end of the project phases.

3. Collecting and studying reports on other similar projects - this provides the possibility of learning from the other projects' and other teams' experiences; in that sense, a process database is essential for an organization that wants to go higher than Level 2 on the CMM (Capability Maturity Mode) level, engineering management depends on measurements, and their proper use, and this data base is to be regarded as an organizational asset, and it is to be properly managed.

4. Top management commitment-if top management does not play a strong, active role in the project from initiation through implementation, then all other risks and issues may be impossible to address in a timely manner.

5. Failure to gain user commitment - when the users are actively involved in the requirements determination process, it creates a sense of ownership, thereby minimizing the risk that the end-user expectations will not be met and that the system will be rejected.

6. Timeliness of additional user requirements - it is essential to have the users involved in the development process from the beginning to the end; however, it is highly preferable to have the requirements frozen at a certain point in development.

7. Familiarity with technology - the higher the organizations experience with application languages, technology databases, hardware, and operating systems, the lower the risk in the project.

8. Insufficient/inappropriate staffing - the risk of failing to provide adequate staffing throughout the project can be mitigated by using disciplined development processes and methodologies to break the project down into manageable chunks, and developing contingency plans.

9. The degree of structure in the project's outputs - it is negatively correlated with the risk of the project.

In the context of the Unified Process of software development, it is adopted that one can never fully eliminate risks; at best, one can manage them [Booch, 1996], [Jacobson et al., 1999]. For that reason, the Unified Process stresses the need to drive software development as an architecture-centric activity. Architecture-centric approach forces the risk factors to emerge early in the development process and make the process simultaneously risk-driven – when the risk factors are identified early, managers can take steps to mitigate them. Experienced software project managers recommend to maintain a running list of project's top ten risk factors and use that list to drive each release [Boehm, 1981].

2.3.11 Standards

There are two major aspects of the term “Standards" in software development. One of them is that of using widely accepted standards under the assumption that they embody “the common body of knowledge and accepted state of industry best practice". Such standards include universally recognized control frameworks for software process control and improvement. Some examples include ISO 9000, ISO 12207, TickIt, PSP, SPICE and Boot strap. They provide a basis for defining systematic activities, roles, and tasks that can be carried out in software development, independent of individual projects, companies, or designers. Furthermore, they make possible to understand and manage all of the diverse

forms of software activity from the standpoint of a single framework. Software project managers should understand and apply these standards and frameworks as points of reference in software development, in order to ensure that quality is being designed and built into the products [Shoemaker, D., Jovanovic, 1999].

2.3.12 Best Practices

Not every practice of software development can be standardized, yet many of them have proved to be useful in a number of projects and organizations. Most of such practices come from experience, and it is extremely beneficial for every project that the project manager and the members of the development team are knowledgeable of as many such practices as possible. The following examples illustrate the idea of using such practices in software project management.

1. Communication problems arising when a distributed team is developing software must be handled with special care. A kick-off meeting must be held face-to-face, and all the developers, partners, and contractors must attend. All product deliverables must be clearly defined in the very beginning [Haywood, M, 2000].
2. Top-down approach to software analysis and design should be enforced in software development projects when the application domain is numerically intensive, such as signal processing, pattern recognition, and real-time control [Fayad, 1996].
3. Productivity of a newly assembled development team should be calibrated in a pilot project. A small pilot project gives project manager the possibility of gaining a rough model of performance for every team member and for the team as a whole before the real work on a large application begins [Booch, G, 1996].

2.4 Conclusion

Software project management issues described in this chapter represent the core of project manager's toolbox for leading the project to successful completion. It is important to note that there are many more interesting topics and practices of software project management, other than those covered here. Due to the enormous expansion of information technology, such practices continue to grow. Again, if a certain issue specific to software development is not addressed in this chapter it doesn't mean that it is not important - it is because it didn't satisfy some of the selection criteria stated in the end of the Introduction.

For example, an important issue for every software project manager is contingency planning. It relates to the effect of strikes by personnel, fire, flooding, earthquakes, each of which can have a rather specific effect on the software process - copies of work products have to be kept off site, and be easily accessible. However, contingency planning doesn't get that much space in the relevant software engineering journals and magazines, or in the publications of the relevant committees associations.

However, it is important to stress that even if all of the important practices and issues could be briefly covered here, software project management in reality requires a more detailed insight into the practices themselves and many case studies, as well as a lot of experience, judgment, and intuition. Experienced managers always take the issues and practices reported in the open literature just as rough guidelines and adapt them to the context of their current project. This suggests an important general rule of thumb: best practices of software project management are always those that can be applied to the system being built, the technology the developers use, and the organization that develops the system. In this way project management manages software metrics and other aspects.

Software project measurement

Introduction

"When you cannot measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: It may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science." [Lord Kelvin 1824-1904]

3.1 Definitions of software metrics

Intuitively one could guess that "software metrics" is involved with numbers and measuring different aspects of software. But to structure our own minds, and to give a more precise idea of what we mean by "software metrics", we need a definition. If we turn to the literature we can find several such definitions, which give more or less the same interpretation of the term.

[Goodman ,1993] defines software metrics as "the continuous application of measurement-based to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve the process and its products" . As one can understand, this definition covers a wide field of application, but the main focus is on improving the software process and all the aspects of the management of that process.

The main situation for use of software metrics is in decision making, which is emphasized by [Grady, 1992]: "Software metrics are used to measure specific attributes of a software product or software development process, they help us to make better decisions". This definition also pinpoints one of the problems of software development today: the lack of information for predicting and evaluating software projects.

Some of the authors make a distinction between Software measurement and software metrics. They mean that the terms measure and measurement are more mathematical correct when discussing the empirical value of different measured objects in the software development process. Further they establish that a metric is a criterion used to determine the difference or distance between two entities, for example the metric of Euclid that measures the shortest distance between two points. These authors claim that the use of the term software metrics is built on a misunderstanding of the correct meaning of it, and that software measurement should be used instead [Zuse, 1991].

Yet another aspect of software metrics that is often emphasized when defining the term, is that it can be applied during the whole development lifecycle, from initiations, when costs must be estimated, to monitoring the reliability of the end product in the field, and the way that the product changes over time with enhancement. Used correctly and consistently a project can therefore benefit from the software metrics during all stages of the software development [Fenton & Pfleeger, 1997].

Sometimes one can come across attempts to classify different types of software metrics. One such distinction is made between primitive and computed metrics. Primitive software metrics are directly measurable or countable, such as counting lines of code. Computed software metrics use some kind of mathematical formula to calculate the value of an attribute of a software project, such as the productivity measure of non-comment source statements per working month [Zuse, 1991].

To sort out the measurements that are used for high-level decision-making the term "global metrics" is often used. This expression is most often used when referring to metrics, which is giving as a result information about size, product, and development process quality that are of interest to the managers of a software development or maintenance activity. This notion implicates that there could also be a such thing as a "local metrics", which is tied up with a more limited practice of software metrics during one phase of the product life cycle, for example code measures for a separate module. These small-scale metrics, usually called "phase metrics", could be combined for a high-level use, and according to the definition, phase metrics are in this way the constituent parts of the managerial global metrics [Goodman, 1993].

This discussion has led us to a definition of software metrics that we are going to use in this report. "**Software metrics are defined as the practice of measuring different attributes of the software development process and products in order to get useful and relevant information for efficient management during the entire development process**"

My view of software metrics is goal-oriented (or top-down). The primary problem for me is not what kind of metric(s) should be used, but rather what the goal of the measurements is and which kind of information project management would like to have. Only when we know the context of the metrics, the purpose of it and the people interested in it, we can correctly identify and evaluate the applicable measurements.

3.2 Why software metrics?

Now that we have a more established idea of what software metrics is, we also need to ask ourselves if and why software metrics matters. Why do we need to measure software? One way to answer this question is to identify the problems that could arise if we do not use software metrics in our projects. In this research

I have found three groups of difficulties for developers and managers, who do not have a notion of software metrics:

1. Developers and Managers cannot set up measurable goals for their software products, since they do not know if they have reached them. For example, they can promise that this or that product should be user-friendly, reliable and easy to maintain, but as long as they do not clearly and objectively specify what they mean by these terms they do not know if they have met their goals. Tom Gilb (1988) has summarized this in his Principle of Fuzzy Targets. This principle says: *Problems without clear goals will not achieve the goals clearly* [Tom Gilb, 1988].

2. For most projects it is rather easy to establish the total cost, but it is harder to distinguish the costs at different stages of the software development process from each other, for example the cost of design from the cost of coding or testing. One reason for many complaints from the customers is also the failure to give a correct estimate of cost. If the managers cannot measure the components of cost it is almost impossible to control the total cost, and consequently hard to give an accurate quotation to the customer [Bache & Bazzana, 1994].

3. Finally, developers and managers fail to quantify or predict the quality of the products they produce. Thus, if the customer want to know how reliable a product will be, or how much work will be needed to change the product, they cannot give the answer. The result of this is that the customer, lacking of valuable information, that perhaps other companies can supply recognizes the risks of choosing their product and therefore purchases a substitute [Fenton & Pfleeger, 1996].

Based on this inventory of software development pitfalls we can list three basic activities for which measurements are important. First, we can identify measures, which are used to *Understand* what is happening during the different stages of development and maintenance. Through measurements we can see clearly the relationships among activities, which factors influence the development process and how they can be influenced. Second, software metrics can help us *control* the activities in our projects. When we understand the relationships, we can use our goals and baselines to try to predict what will happen and make changes to processes and products in order to meet our goals. Third, measurement supports the activity to improve our processes and products. For example, by sorting out those parts of the project that do not meet our quality requirements, and deposit more resources to monitoring these parts, we can improve our overall quality.

3.3 Recipients and use of software metrics information

Here I classified the recipients as managers, engineers and customers. I have already shortly described which members of the software development team that may be interested in the information we get from software metrics. I choose,

perhaps somewhat roughly, to separate them into two groups: managers and engineers. However, I acknowledge that there could be many different kinds of managers (economical managers, software quality managers, technical managers, staff manager etc.) and engineers ("design engineers", "code-generating engineers", "testing and verifying engineers" etc.). If we leave this distinction out of consideration, we can identify a wide range of useful information for these two groups. In addition, the interests of customers/users/clients can be satisfied by software metrics information.

In Perspective of Managers I discuss it in the relation between project management and Measurement. See section 2.2

3.3.1 Engineers

One obvious advantage of software metrics, which we already have touched upon, is that different attributes of the software process are objectively measurable and can therefore be expressed in numbers. The developers then have the opportunity to translate the requirements from words to testable entities. For example, one that states that the lower limit for mean time to failure is 15 elapsed hours of CPU time can replace the reliability requirement.

If we measure the number of faults during the different phases of the software product life cycle, and build a database on this information, it is also possible to set up models of expected detection ratios. These models can help us to evaluate future testing efforts and decide whether they have been successful or not, i.e. if it is probable that we have found all the faults. The measurement of the characteristics of the products and processes can also tell us whether we have met our quality standards and our quality goals. For instance, if we have specified that the software should not contain more than 20 failures per time unit, we can easily control this during the test phase. But software metrics are not only useful for a retrospective view on the development process; we can also use it for predicting attributes of the future products and processes, such as probable system size, maintenance problems and software reliability [Fenton & Pfleeger, 1996].

3.3.2 Customers

Software metrics is of obvious interest for the people who will use the system produced. Firstly, it is important for them to have information about quality, cost, time consumption etc., to be able to compare different products or companies with each other. When this information is available for all the alternatives that a customer can make a more rational and well-founded choice. On the other hand, if the information is not available for some alternatives, the customer is more likely to choose those products from companies that have bothered to gather the information.

Secondly, information from a software metrics' program is meaningful for the users when they would like to evaluate a project developed or product produced

by another company (outsourcing). Measurements can then be used to assess the quality of a software system. The question is whether the system have the reliability, maintainability, usability, efficiency etc., that we would like it to have. Software metrics can also be used to discover if a project has not performed as good as we have hoped relative to the cost of the project, i.e. those projects with too low productivity [Möller & Paulish, 1993].

Evidently, there are problems associated with companies giving this type of information to its customers. If competitors can capture this information they can use it for own purposes, for example by discrediting the measured company in marketing. "Bad" figures of e.g. fault density or programmer productivity can do a great harm to a software company, and thus it is likely that it will retain the information if it can suspect that it will be public if it is released to the customers.

3.5 Models in Software metrics

Software metrics includes many types of models and measures used in the situations described above. There are many proposals in the literature of how to classify measures for different areas of the software development[Möller & Paulish, 1993; Fenton & Pfleeger, 1996; Ohlsson, 1996; Grady, 1992], and I have tried to summarize them in the following categories:

All these functions are done and managed by project management that organize, select the models at different stages to keep track of the project and to make correct decision.

1.***Cost and effort estimation models****:* The aim of these models is to predict total cost of a software development project mainly at the requirement stage, but also to track the costs during the whole product life cycle. An example of such a model is Albrecht's Function Points model. The models often share the approach of effort expressed as a function of one or more variables (for example size, capability of the developers and level of reuse). Size is usually computed by counting Lines of Code.

2. ***Productivity models and Measures****:* When combining measures of size and effort or cost there is the possibility to reach a productivity measure. Based on the collection of productivity data from finished projects, managers can also build models for assessing and predicting staff productivity in future projects. These models and measures are on different levels of sophistication from the traditional ones, that divides size by effort, to ones that take more factors into consideration, such as quality, functionality and complexity.

3***. Quality Measures and Models***: Productivity cannot be viewed in isolation. The speed of production is meaningless if the product is of inferior quality. This discovery has led software engineers to develop models of quality whose measurements can be combined with those of productivity models.

4. ***Reliability Models:*** Most quality models include reliability as a factor, but the need, above all generated from the customers, for reliable software has led to the specialization in reliability modeling and prediction. Reliability models are usually statistical models for predicting mean time to failure or expected failure interval.

5. ***Structural and Complexity Metrics:*** Some quality attributes, like reliability and maintainability, are not measurable until the operational version of the code is available. To be able to predict which modules in a system that are less reliable than others, different predictive theories have been established to measure structural attributes of the software to support quality assurance, quality control and quality prediction. Examples of such theories are Halstead's measures of effort, difficulty, volume and length, as well as McCabe's cyclomatic number.

3.6 Conclusions

Although the discipline of software metrics is rather young one, it influences many areas of the software development process. In this chapter I have tried to give a defined the basic concept, explained in which areas they are applied and for whom it is useful. I developed our own definition of software metrics, based on the literature, and I established that software metrics is the practice of measuring different attributes of the software development process and products in order to get useful and relevant information for efficient management during the entire development process.

Three main reasons for using software metrics can be identified. Firstly, if we are not able to measure our products and processes, we cannot set up measurable goals, since we do not know if we have reached them. Secondly, we may want to distinguish costs at different stages of the software development process from each other. This is possible with software metrics. Finally, the customers will not choose our products if we are not able to quantify or predict the quality of the products we produce.

I also distinguish three main recipients of software metrics information they are managers, engineers and customers. ***Managers*** use it mainly for controlling and predicting the cost of projects, but also to evaluate the quality, and calculate the effects of a more cost-effective solution. The ***Engineers*** are more concerned with the quality of the system, and especially if the system is meeting the requirements. ***Customers*** are also very interested in the quality of the system, but mainly from their own viewpoint and according to their own requirements. Problems with giving this kind of information to the public can also be recognized.

Finally I tried to categorize different models in software metrics, cost and effort estimation, productivity models and measures, quality models and measures, reliability models, performance evaluation and models, and finally structural and complexity metrics for successful project management.

Software cost estimation

4.1 Introduction

Generally there are many things we can measure in a project like schedule, cost, quality, projects similarity, effort etc. There are many methods and models proposed till now. Effective software project estimation is one of the most challenging and important activities in software development. Proper project planning and control is not possible without a sound and reliable estimate. As a whole, the software industry doesn't estimate projects well and does not use estimates appropriately. We need to put more effort for improving the situation and models proposed for to measure cost and schedule [Douglis, Charles, 1998].

Software cost estimation: It is a process in which an organization uses certain set of techniques and procedures to estimate cost. For a success in project, project management should concentrate on activities like planning, controlling, improving and organizing which are very difficult [Douglis, Charles, 1998]. We face many problems if we do not estimate the measurable factors. We have problems of under-staffing (resulting in staff burnout), under-scoping the quality assurance effort (running the risk of low quality deliverables), and setting too short schedules (resulting in loss of credibility as deadlines are missed). For those who figure on avoiding this situation by generously padding the estimate, over-estimating a project can be just equally bad for the organization! If we give a project more resources than it really needs without sufficient scope controls, it will use them. The project is then likely to cost more than it should (a negative impact on the bottom line), take longer to deliver than necessary (resulting in lost opportunities), and delay the use of your resources on the next project [Douglis, Charles, 1998].

This elucidates us the importance of cost estimation. There are many methods and models for the cost estimation:

- algorithmic methods
- estimating by analogy
- expert judgment method
- top-down method
- bottom-up method...

	Expert Judge Method	**Anology Method**	**Top-Down Method**	**Bottom up Method**	**Algorithmic Method**
Definition	It is a method that involves consulting with software cost estimation experts to use their experience and understanding of the proposed project to arrive at an estimate of its cost.	In this method comparison is done between proposed projects and previously completed similar projects where the project development information is known.	It is also called Macro Model. In this method cost estimation is derived from the global properties of the software project, and then the project is partitioned into various low-level components.	In this method the cost of each software components is estimated and then combines the results to arrive at an estimated cost of overall project.	This method is effective and is designed to provide some mathematical equations to perform software estimation.
Advantages	It is empirical	It is based on actual experience of projects	This method is efficient and system level view	Detailed and stable	1) It is Objective and repeatable. 2) It has modifiable and analyzable formulas.
Disadvantages	It is subjective	Difficult to ensure the degree of representative between previous projects and new one	Too rough	Overlook many of the system-level costs, inaccurate, more time-consuming	1)Unable to deal with exceptional conditions. 2)Some experience and factors can not be easily quantified

Figure 4.1: Methods For Cost Estimation

No method is necessarily better or worse than the other, in fact, their strengths and weaknesses are often complementary to each other. The models that are used mostly now-a days are derived from algorithmic method. For example COCOMO & COCOMO II, SLIM, ESTIMACS and SPQR/20.

These are some of the tools for the cost estimation that are more suitable for large companies. These are some models existing for Software cost estimation [Vidger, M.R. and Kark, 1994].

- ACEIT
- COCOMO II
- Construx Estimate
- COSMOS

- COSTAR
- Cost Xpert
- Estimate Pro
- PRICE-S
- SEER-SEM
- SLIM ESTIMATE

In the market COCOMO II is the model, which is being used most, due to its flexibility than other models, Now we will see a detail description of it to have a clear idea and to evaluate good measures for Small scale organizations.

I have considered COCOMO II and SLIM models to evaluate and suggest suitable measures for the small-scale enterprises. To have a clear idea about the conclusions in the next chapter it is essential to have a look at these models in brief.

4.2 COCOMO Models

The original COCOMO Model has now been superceeded by COCOMO II. Firstly COCOMO means Cost Constructive Model, However a brief review of the original COCOMO model provides insight into the evolution of software estimation approaches. The hierarchy of COCOMO models takes the following form:

Model 1. The Basic COCOMO model is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code (LOC).

Model 2. The Intermediate COCOMO model computes software development effort as a function of program size and a set of "cost drivers" that include subjective assessments of product, hardware, personnel and project attributes.

Model 3. The Advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

4.2.1 COCOMO I Model

To illustrate the COCOMO model, here is the brief overview of the Basic and Intermediate versions. For a more detailed discussion, the reader is urged to study [Vidger, M.R. and Kark, 1994].

The COCOMO models are defined for three classes of software projects. Using Boehm's terminology these are:

Organic mode: Relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements (e.g., a thermal analysis program developed for a heat transfer group).

Semi-detached mode: An intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements (e.g., a transaction processing system with fixed requirements for terminal hardware and data base software).

Embedded mode: A software project that must be developed within a set of tight hardware, software and operational constraints (e.g., flight control software for aircraft).

The Basic COCOMO equations take the form:

$E = a_b \, KLOC^{\,b_b}$; $D = c_b \, E^{\,d_b}$

Where *E* is the effort applied in person-months, *D* is the development time in chronological months and KLOC is the estimated number of delivered lines of code for the project (express in thousands). The coefficients a_b and c_b and the exponents b_b and d_b are shown in Table 4.2

Figure 4.2: Basic COCOMO Model [Boehm Barry W, 1981].

Software Project	**a_b**	**b_b**	**c_b**	**d_b**
organic	2.4	1.05	2.5	0.38
semidetached	3.0	1.12	2.5	0.35
embedded	3.6	1.20	2.5	0.32

Table 4.3: Cost driver attributes [Boehm, Barry W, 1981].

Product attribute	**Hardware attributes**	**Personal attributes**	**Project attributes**
required software reliability	run-time performance constraints	analyst capability	use of software tools
size of application data base	memory constraints	software engineer capability	application of software engineering methods
complexity of the product	volatility of the virtual machine environment	applications experience	required development schedule
----------	required turnaround time	virtual machine experience	
----------		programming language experience	

The basic model is extended to consider a set of "cost driver attributes" that can be grouped into four major categories as given in the Table 4.3.

Each of the 15 attributes is rated on a 6 point scale that ranges from "very low" to "extra high" (in importance or value). Based on the rating, an effort multiplier is determined from tables published by Boehm, and the product of all effort multipliers results is an *effort adjustment factor* (EAF). Typical values for EAF range from 0.9 to 1.4.

The intermediate COCOMO model takes the form:

$E = a_i \text{ KLOC}^{b_i} \text{ x EAF}$

where *E* is the effort applied in person-months and *KLOC* is the estimated number of delivered lines of code for the project. The coefficient a_i and the exponent b_i are given in Table 4.4

Table 4.4: INTERMEDIATE COCOMO MODEL [Boehm, Barry W, 1981].

Software Project	$\mathbf{a_i}$	$\mathbf{b_i}$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

COCOMO represents a comprehensive empirical model for software estimation. However, Boehm's own comments [Boehm, Barry W, 1981] about COCOMO (and by extension all models) should be heeded:

Today, a software cost estimation model is doing well if it can estimate software development costs within 20% of actual costs, 70% of the time, and on its own turf (that is, within the class of projects to which it has been calibrated) This is not as precise as we might like, but it is accurate enough to provide a good deal of help in software engineering economic analysis and decision making. This model was based on the trends used in software engineering during the 70-80's and as we all know that their had been a great change in the field of software i.e. it has matured to a great extent therefore large group of scientist have done a long research and came up with a solution to meet up the present standards of the software. And they named it as COCOMO II. It's an extension of the above model [Boehm, B.W., Abts, C., Clark B, Devnani Chulani, 1997].

4.2.2 COCOMO II

It is a model that allows one to estimate the cost, effort, and schedule when planning a new software development activity. It consists of three sub models, each one offering increased fidelity along the project planning and design process. Listed in increasing fidelity, these sub models are called the Applications Composition, Early Design, and Post-architecture models. Until recently, only the

last and most detailed sub model, Post-architecture, had been implemented in a calibrated software tool [Boehm Barry W, 1981]. The initial definition of COCOMO II and its rationale are described here. The definition will be refined as additional data are collected and analysed. The primary objectives of the COCOMO II effort are:

1. To develop software cost and schedule estimation model tuned to the life cycle practices of the 1990's and 2000's.

2. To develop software cost database and tool support capabilities for continuous model improvement.

3. To provide a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle costs and schedules.

In priority order, these needs were for support of project planning and scheduling, project staffing, estimates-to-complete, project preparation, replanning and rescheduling, project tracking, contract negotiation, proposal evaluation, resource levelling, concept exploration, design evaluation, and bid/no-bid decisions. For each of these needs, COCOMO II will provide more up-to-date support than the original COCOMO and Ada COCOMO predecessors [Boehm Barry W, 1981].

Strategy : The four main elements of the COCOMO II strategy are:

1. Preserve the openness of the original COCOMO;
2. Key the structure of COCOMO II to the future software marketplace sectors described above;
3. Key the inputs and outputs of the COCOMO II sub models to the level of information available;
4. Enable the COCOMO II sub models to be tailored to a project's particular process strategy.

COCOMO II follows the openness principles used in the original COCOMO. Thus, all of its relationships and algorithms are publicly available. The interfaces are designed to be public, well defined, and parameterised, so that complementary pre-processors (analogy, case-based, or other size estimation models), post-processors (project planning and control tools, project dynamics models, risk analysers), and higher-level packages (project management packages, product negotiation aids), can be combined straightforwardly with COCOMO II. COCOMO II includes 3 different stages:

The Application Composition Model: Suitable for projects built with modern GUI-builder tools.

The Early Design Model: Here in this model we can achieve rough estimates of a project's cost and duration before the determination of it's entire architecture. It uses a small set of new Cost Drivers, and new estimating equations. Based on Unadjusted Function Points or KSLOC.

The Post-Architecture Model: This is the most detailed COCOMO II model. This model is used after the development of overall projects architecture. It has new cost drivers, new line counting rules, and new equations.

Effort estimation: In COCOMO II effort is expressed as Person Months (PM). A person month is the amount of time one person spends working on the software development project for one month. This number is exclusive of holidays and vacations but accounts for weekend time off. The number of person months is different from the time it will take the project to complete; this is called the development schedule. For example, a project may be estimated to require 50 PM of effort but have a schedule of 11 months [Boehm, Barry W, 1997].

Before going to the equations we have to consider scaling factors, cost drivers and source lines of codes.

Scale drivers: In the COCOMO II model, some of the most important factors contributing to a project's duration and cost are the Scale Drivers. The Boehm group set 5 Scale Drivers to describe the project; these Scale Drivers determine the exponent used in the Effort Equation. The 5 Scale Drivers are:

1. Precedentedness
2. Development Flexibility
3. Architecture / Risk Resolution
4. Team Cohesion
5. Process Maturity

Whereas COCOMO is reasonably well matched to custom, build-to-specification software projects, COCOMO II is useful for a much wider collection of techniques and technologies. COCOMO II provides up-to-date support for business software, object-oriented software, and software created via spiral or evolutionary development models, and software developed using commercial-off-the-shelf application composition utilities [Boehm et al, 1997]. COCOMO II includes the Application Composition model (for early prototyping efforts) and the more detailed Early Design and Post-Architecture *models* (for subsequent portions of the lifecycle). For detail description see [Boehm's 1999].

4.3 SLIM (Software Life-cycle Model)

Putnam developed a constraint model called SLIM to be applied to projects exceeding 70,000 lines of code. Putnam's model assumes that effort for software projects is distributed similarly to a collection of Rayleigh curves. Putnam suggests that staffing rises smoothly during the project and then drops sharply

during acceptance testing. The SLIM model is expressed as two equations describing relation between the development effort and the schedule. The first equation, called the *software equation*, states that development effort is proportional to the cube of the size and inversely proportional to the fourth power of the development time. The second equation, the *manpower-buildup* equation, states that the effort is proportional to the cube of the development time.

The Norden-Rayleigh Curve: The Norden-Rayleigh curve represents manpower as a function of time. Norden observed that the Rayleigh distribution provides a good approximation of the manpower curve for various hardware development processes [Pillai, K. and Sukumaran Nair, V, 1997]. SLIM uses separate Rayleigh curves for design and code, test and validation, maintenance, and management. A Rayleigh curve is shown in Figure 1.

Figure 4.4: A Rayleigh curve [Pillai, K. and Sukumaran Nair, V, 1997].

Development effort is assumed to represent only 40 % of the total life cycle cost. Requirements specification is not included in the model. Estimation using SLIM is not expected to take place until design and coding. Several researchers have criticized the use of a Rayleigh curve as a basis for cost estimation [Pillai, K. and Sukumaran Nair, V, 1997]. Norden's original observations were not based in theory but rather on observations. Moreover his data reflects hardware projects. It has not been demonstrated that software projects are staffed in the same way. Software projects sometimes exhibit a rapid man power build up which invalidate the SLIM model for the beginning of the project.

The Software Equation: Putnam used some empirical observations about productivity levels to derive the software equation from the basic Rayleigh curve formula [Fenton, N.E. and Pfleeger, 1997]. The software equation is expressed as:

Size = CE^1/3(t^4/3) (Size = Software product size)

Where *C* is a technology factor, *E* is the total project effort in person years, and *t* is the elapsed time to delivery in years. The technology factor is a composite cost driver involving 14 components. It primarily reflects:

- Overall process maturity and management practices
- The extent to which good software engineering practices are used
- The level of programming languages used
- The state of the software environment
- The skills and experience of the software team
- The complexity of the application

The software equation includes a fourth power and therefore has strong implications for resource allocation on large projects. Relatively small extensions in delivery date can result in substantial reductions in effort [Pressman, R.S, 1997].

The Manpower-Build up Equation: To allow effort estimation, Putnam introduced the manpower-build up equation:

D = E / t^3

Where *D* is constant called manpower acceleration, *E* is the total project effort in years, and *t* is the elapsed time to delivery in years. (Years is the unit of dimension here)The manpower acceleration is 12.3 for new software with many interfaces and interactions with other systems, 15 for standalone systems, and 27 for re implementations of existing systems. Using the software and manpower-build up equations, the effort is calculated as [Putnam and Myers , 1992].

E = (S / C)^9/7 (D^4/7)

This equation is interesting because it shows that effort is proportional to size to the power 9/7 or ~1.286, which is similar to Boehm's factor which ranges from 1.05 to 1.20.It uses linear programming to consider development constraint on both cost and effort [Fenton, N.E. and Pfleeger, S.L, 1997].

4.4 Conclusion

In this Chapter we have seen two estimation tools, which are usually used in Estimating Cost of Large Scale organisation. These tools have been chosen randomly, the factors, which this tools use in large organisations will be applied to the properties of small-scale organisation and suitable measures are suggested in Chapter 5 by comparing both the tools and appropriate factors for small-scale organisation.

Project management measures for SE's

5.1 Introduction

In this chapter there is a classification of small organizations in to 3 three different categories based on the number of employees and number of the products under development. Then I have considered the measurable factors of software development in large-scale organizations like size, effort, cost quality and maintainability. I have analyzed how these factors are measured in Large-scale organizations and according to the observation and properties of small organizations I have suggested the suitable measures for SME's. I have also given suggestions when estimation is done in a new organization.

5.2 Classification of Small organizations

As part of the research project QMSE (quality management for small enterprises) conducted at Umea University, Sweden. According to Orci and laryd small organisations are classified in three categories based on number of employees and the number of the products under development [Orci and Laryd, 2002]. They are

Table 5.1 Classification of organisation [Orci and Laryd, 2002].

Category	Number of employees
XXS (eXtra eXtra Small)	1-2 employees and 1 product
XS (eXtra Small)	3-15 employees & several products/versions
S (Small)	16-50 employees & several products/versions
Medium	50-250 employees & several products/versions
Large	250 and more employees & several products/versions

Here the administrative staff is included in number of employees. This is how organisations are divided on the basis of employees according to Tertu and Laryd [Tertu orci and Laryd, 2002]. In general large-scale enterprises has more than 250 employees, they develop many projects simultaneously, they deal with big budgets. Where in medium enterprises numbers of employees are less than 250, medium enterprises also develop several products simultaneously but comparatively less than the large enterprises and it deals with medium budgets. And in small enterprises number of employees is less as shown in the above table with low budgets.

Now we have the idea of both large scale companies and small scale companies, as my thesis demands I tailor and evaluate different methods for SE's in the below section.

5.3 Evaluation of different measures for measurable factors in SE's

Either it is small companies or big companies the first and foremost things we consider are the profit and the quality of the software product. For the profit of the organisation, the important factor is cost. We shall see every aspect of cost which is measurable and measurable factors in software development, and how it is done SE's. Software project measurement regardless of the size deals with the following factors.

- It helps us to estimate cost and effort devoted to project.
- It helps us to determine quality of software.
- It helps us to predict maintainability of software.
- It helps us to validate the best practices of software development.

In this section we will see what are the suitable measures for SE's and how they are useful. Here the classification of organisations is done by Terti orci and Laryd is considered. As the software project measurement supports the above statements, in this section an evaluation and tailoring of suitable of measures for SE's is presented.

5.4 Size

In order to estimate effort we require size of software. Size is a primary cost factor in most models. There are two common ways to measure software size: *lines of code* and *function points*.

5.4.1 Lines of Code

The most commonly used measure of source code program length is the number of lines of code (LOC) [Boehm, B.W, 1997]. The abbreviation NCLOC is used to represent a non-commented source line of code. NCLOC is also sometimes

referred to as effective lines of code (ELOC). NCLOC is therefore a measure of the uncommented length.

The commented length is also a valid measure, depending on whether or not line documentation is considered to be a part of programming effort. The abbreviation CLOC is used to represent a commented source line of code [Boehm, B.W, 1997]. By measuring NCLOC and CLOC separately we can define:

Total length (LOC) = NCLOC + CLOC

Where KLOC is used to denote thousands of lines of code.

5.4.2 Function Points

Function points (FP) measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an unadjusted function point count (UFC). Counts are made for the following categories [Boehm, B.W, 1997].

- *External inputs* – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)
- *External outputs* – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
- *External inquiries* – interactive inputs requiring a response
- *External files* – machine-readable interfaces to other systems
- *Internal files* – logical master files in the system

Once these data has been collected, a complexity rating is associated with each count as shown in table 5.2.

Table 5.2: Function point complexity weights [Pressman, 1997].

Weighting Factor			
Item	**Simple**	**Average**	**Complex**
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

Each count is multiplied by its corresponding complexity weight and the results are summed to provide the UFC. The adjusted function point count (FP) is calculated by multiplying the UFC by *a* technical complexity factor (TCF). Components of the TCF are listed in Table 5.3

Table 5.3 Components of the technical complexity factor [Pressman, 1997].

F1	Reliable back-up and recovery	F2	Data communications
F3	Distributed functions	F4	Performance
F5	Heavily used configuration	F6	Online data entry
F7	Operational ease	F8	Online update
F9	Complex interface	F10	Complex processing
F11	Reusability	F12	Installation ease

Each component is rated from 0 to 5, where 0 means that component has no influence on the system and 5 means the component is essential [Pressman, 1997]. The TCF can then be calculated as:

TCF = 0.65 + 0.01(SUM (Fi))

The factor varies from 0.65 (if each *Fi* is set to 0) to 1.35 (if each *Fi* is set to 5) [Boehm, B.W, 1997] .The final function point calculation is:

FP = UFC x TCF

We can also measure size by analogy. Having done a similar project in the past and knowing its size, you estimate each major piece of the new project as a percentage of the size of a similar piece of the previous project. Estimate the total size of the new project by adding up the estimated sizes of each of the pieces. An experienced estimator can produce reasonably good size estimates by analogy if accurate size values are available for the previous project and if the new project is sufficiently similar to the previous one [Boehm, B.W, 1997].

5.4.3 Estimating size in XXS, XS, S

For a XXS company the concept of lines of code is suitable as it suits in large-scale organisation. And when function points are consider they are more complicated than the LOC because this is a small organisation with less number of employees than in large scale organisation and moreover there are number of factors which have to be considered.

In XS and S companies measuring size is performed in the same way. If they have done the similar project before then the method of analogy is more suitable, if they had not develop the similar project then they should follow the same as XXS but the function points suits more here due to increase in the employees. The concept of Lines of Codes is suitable for both large scale organisation as well as in XXS companies.

5.5 Effort

In general effort is estimated after finding the size. After calculating the size it's easy to calculate effort. Once we have an estimate of the size of our product, we can derive the effort estimate. This conversion from software size to total project effort can only be done if we have a defined software development lifecycle and development process that we follow to specify, design, develop, and test the software. A software development project involves far more than simply coding the software – in fact, coding is often the smallest part of the overall effort. Writing and reviewing documentation, implementing prototypes, designing the deliverables, and reviewing and testing the code take up the larger portion of overall project effort. The project effort estimate requires we to identify and estimate, and then sum up all the activities we must perform to build a product of the estimated size [Vidger M,R & A.W Kark, 1994]. There are two main ways to derive effort from size, the first approach uses the organization's own historical data to determine how much effort previous projects of the estimated size have taken. This, of course, assumes that:

- Our organization has been documenting actual results from previous projects,
- That we have at least one past project of similar size (it is even better if we have several projects of similar size as this reinforces that we consistently need a certain level of effort to develop projects of a given size), and
- That we will follow a similar development lifecycle, use a similar development methodology, use similar tools, and use a team with similar skills and experience for the new project.

The second approach to estimate effort is to use algorithmic model. If we don't have historical data from our own organization because we haven't started collecting it yet or because our new project is very different in one or more key aspects, we can use a mature and generally accepted algorithmic approach such as Barry Boehm's COCOMO model or the Putnam Methodology to convert a size estimate into an effort estimate. These models have been derived by studying a significant number of completed projects from various organizations to see how the project sizes mapped into total project effort. These "industry data" models may not be as accurate as our own historical data, but they can give us useful effort estimates [Goether, Wolfhart B., Elizabeth K. Bailey, Mary B. Busby, 1992]. Both models are discussed in the chapter 4.

5.5.1 Estimating effort in XXS, XS, S companies

Two cases are considered in XXS

1. New organisation
2. Old organisation

When we consider the XXS as new then it hasn't any historical data as a consequence it cannot use and predict effort of the software project. And

moreover its running only one product that too it is a new one which is obvious that it doesn't have any similar projects therefore we can't use similar development cycle, similar methodology, similar tools. Therefore the first approach it's not suitable to an XXS and the second approach of automated tools like COCOMO models and SLIM model are more suitable because this is a new organisation with out any experience, there is a possibility of forgetting many aspects . When doing effort estimation if we use these tools they will not allow us to forget any factors and we will have a good estimation. More specifically in COCOMO organic mode and semidetached models are more suitable because it's a new organisation with a less experience and with out historical data. These modes exactly fit the XXS features. If we consider XXS as an old organisation then the method of analogy is suitable here because it has previous historical data which can be used to predict methodology, tools to be used, development cycle to be followed etc. Since it is a old organisation they will have some experience in doing effort with historical data therefore if they can manage estimating effort by historical data then it would be more advantageous then using COCOMO and SLIM because these models are expensive.

XS and S companies are probably working on number of products by which they have previous data and with size they can calculate effort in persons-months. And more over they have more employees than XXS companies. If the product is quite different then automated tools like COCOMO II more suitable because the small organisation has many similar functions to perform as large organisations to produce good quality product.

5.6 Schedule

In general the next thing in estimating a software development project is to determine the project schedule from the effort estimate. This generally involves estimating:

1. The number of people who will work on the project.
2. What they will work on (the Work Breakdown Structure).
3. When they will start working on the project and when they will finish (this is the "staffing profile").

Once we have this information, we need to lay it out into a calendar schedule. Again, historical data from our organization's past projects or industry data models can be used to predict the number of people you will need for a project of a given size and how work can be broken down into a schedule [Goether, Wolfhart B., Elizabeth K. Bailey, Mary B. Busby, 1992].

5.6.1 Estimating schedule in XXS, XS, S companies

In XXS two cases are considered

1. New organisation
2. Old organisation.

In a XXS company we have only two persons working on one product. In general by historical data we can say how many people to be appointed. If we consider a new organisation then we don't have any historical data and hence the analogy method is not suitable. In XXS there are only two people working on one product. It all depends on the efficiency and hard work of the employees but they must not have any fixed commitments for the deadlines. If it's an old organisation if it has a historical data then they can follow the previous schedule.

In an XS company according to Orci and Laryd the number of employees in it are 3-15 which is not too less and moreover they are working on number of products, therefore they probably have some experience in doing similar projects with that experience they can predict the number of people accordingly, since it has maximum of 15 employees it is better to make a long time project i.e we should increase the number of months rather increasing persons.

Small organisations S are working on few products with handful of employees, Therefore using historical data and the experience of developing similar projects they can predict number of people and can make the schedule as they are made in large organisations as there is not much difference.

5.7 Cost

In general there are many factors to consider when estimating the total cost of a project. These include labour, hardware and software purchases or rentals, travel for meeting or testing purposes, telecommunications (e.g., long distance phone calls, video-conferences, dedicated lines for testing, etc.), training courses, office space, and so on. Exactly how we estimate total project cost will depend on how our organization allocates costs. Some costs may not be allocated to individual projects and may be taken care of by adding an overhead value to labour rates (-per hour). Often, a software development project manager will only estimate the labour cost and identify any additional project costs not considered "overhead" by the organization. The simplest labour cost can be obtained by multiplying the project's effort estimate (in hours) by a general labour rate ($ per hour). A more accurate labour cost would result from using a specific labour rate for each staff position (e.g., Technical, QA, Project Management, Documentation, Support, etc.). We would have to determine what percentage of total project effort should be allocated to each position. Again, historical data or industry data models are useful [McConnell, Steve, 1996].

5.7.1 Estimating cost in XXS, XS, S

In a XXS company, we have only 2 labours and we don't have any historical data if it is a new organisation. But the effort in hours is multiplied by the cost of the labour per hour (in the own currency). In XXS since we have a only two employees developing only one product then using COCOMO model and SLIM model will be expensive. Therefore it depends on the capability of individuals if it is a new organisation. It would be better if they estimate it slightly higher than

their obtained information from measurement because it's a new organisation with no experience there is a chance of forgetting some factors. If it's a old organisation then they might have some historical data or the similar projects developed with this information they can predict the cost by method of analogy.

In XS and S companies they have number of employees and several products under development. Probably they may have some historical data and would have developed similar projects before, with this knowledge they can roughly estimate the cost by the method of analogy. COCOMO is more suitable here then SLIM if the projects source code size is less than 70,000 lines. If the source code size is more than 70,000 lines then both models will fetch good results. Estimates for small projects are highly dependent on the capabilities of the individuals performing the work. An approach such as Watts Humphrey's Personal Software Process (PSP) [Humphrey 1995] is much more applicable for small project estimation.

5.8 Quality

It is an ambiguous concept, different people view quality in a different way. In the view of the customer quality means the end product is error/bug free one. In the view of the developer, user wants to develop more efficient and reliable program for the project. In the view of the Manager, project must be efficient, cost effective, and reliable, and also customer satisfied. Development of a quality product is the key for customer satisfaction. Software, which does not fulfill the requirements, is an indication of failure. Measuring the quality of the software is a vital step in software engineering [Oulu 94]. We can also say as problems and defects are inversely proportional to quality of software. Software problems and defects are among the few direct measurements in software process and products. Those measurements help us to describe trends in defect or problem discovery, repairs [IEEE 90B].

Now the question is how the quality of software is measured?

Problem and defect measurement are the basis for quantifying several significant software quality attributes, factors and criteria. Software quality attributes of a software product are used to describe and evaluate the quality of the product. A software quality attribute may be refined into multiple levels of sub-attributes. There is also the concept of high-level quality attributes and low-level quality attributes. High-level quality attributes are at a high level of abstraction or generalization that can usually be broken down into sub-attributes. For example, the attribute reliability can be broken down into the sub-attributes Maturity, Fault-Tolerance, and Recoverability. Achieving these more specific sub-attributes will mean achieving the overall attribute of reliability. The sub-attributes can occasionally be again divided into sub-sub-attributes. These forms a tree of attributes starting from the most general (and high-level) attribute at the root and the most specific (and low-level) attributes at the bottom.

Some of the quality attributes measuring software quality are given below [IEEE 90B].

Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy the stated or implied needs of the client.

Reliability: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. 1 - (Errors / lines of code**)**

Usability: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use by a stated or implied set of users.

1- (Labour days to use/labour years to develop)

Efficiency: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

1- (actual utilization/ allocated utilization)

Maintainability: A set of attributes that bear on the effort needed to make modifications to the finished system.

1- 0.1 (average labour – days to fix)

Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another.

1- (effort to transport /effort to develop)

There are many quality attributes, It does not mean that other attributes are not important, I have chosen them randomly to have the clear idea about the quality attributes. These attributes are subdivided into suitability, accuracy, maturity, understand ability, changeability, adaptability etc.

5.8.1 Defects classification

Problems and defects are inversely proportional to the quality of project [IEEE 90B], these are defects we come across in different fields areas of software. They are

Requirements defect: A mistake made in the definition or specification of the customer needs for a software product. This includes defects found in functional specifications; interface, design, and test requirements; and specified standards.

Design defect: A mistake made in the design of a software product. This includes defects found in functional descriptions, interfaces, control logic, data structures, error checking, and standards.

Code defect: A mistake made in the implementation or coding of a program. This includes defects found in program logic, interface handling, data definitions, computation, and standards.

Document defect: A mistake made in software product documentation. This does not include mistakes made to requirements, design, or coding documents.

Test case defect: A mistake in the test case causes the software product to give an unexpected result.

Other work product defect: Defects found in software artifacts that are used to support the development or maintenance of a software product. This includes test tools, compilers, configuration libraries, and other computer-aided software engineering tools. This is the checklist with which we can count the defects and gain a historical data with which the quality can be controlled.

Table 5.4 : Checklist for defects [IEEE 90B]

Software defect	Include	Exclude	Value count	Array count
Requirement defect				
Design defect				
Code defect				
Documentation Defect				
Test case defect				
Other defects				

All these defects are directly proportional to cost, schedule etc Here are the effects I found on cost and schedule due to defects ,

5.8.2 Analysis of quality measurement in XXS,XS,S

Regardless of size the quality of software is important in any organisation, if the quality of the software produced is bad then it is a failure, which results in bad reputation of the company. If the reputation is bad then there will not be any customers. Therefore maintenance of the software quality is important in any organisation. To have a good quality of a software we need measurement, according to [IEEE 90B] quality can be measured by the various defects and with that information quality can be tracked and control. Moreover Cost and schedule are effected by defects.

Effect of defects on Cost: If the defects are found then the rework is necessary to correct the defect, the amount of rework is a significant cost factor in software development and maintenance. The number of problem and defects are the direct

contribution to cost. The staffing must be allotted to correct the flaws which is directly proportional to the cost.

Effects of defects on Schedule: If a fault is detected it must be corrected this require the staff to perform some extra work and hence rescheduling must be done. In this way quality of the software is connected to many aspects of the development process. Finally measurement of the problems and defect are helpful because we can answer how, when, what, who and why? the problem and defects have occurred. This provides good historical data to be used in the further projects. And it also provide insight to methods of detection, prevention and prediction. Finally it keeps cost under the control.

Therefore regardless of the size of the organisation, measurement of the quality is important and the above method described is suitable for any type of organisation.

5.9 Maintainability

Maintainability is an attribute of quality, maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. [Boehm, Barry W, 1997]. Maintainability is ambiguous but the cost of maintenance is high, currently there is no software to measure the maintenance. Despite the subjectivity of any attempt to measure maintainability, great effort has been put into constructing formulas for describing maintainability. Following the opinion that maintainability "is the set of attributes that bear on the effort needed to make specified modifications" [Jones, C, 1996], Rikard Land describes maintainability as a function of directly measurable attributes *A*1 through *A*n, that is:

$$M = f(A1, A2, \ldots, An)$$

A maintainable system must be consistent and simple. However, there are great difficulties in measuring those attributes and weighting them against each other and combine them in a function *f*. Any such attempt is therefore bound to a quite limited context – a particular programming language, organization, type of system, type of project; the skill and knowledge of the people involved must also be considered then drawing conclusions. Typically, maintainability measures are validated using expert judgments about the state of different systems and modules [Coleman D., Ash D., Lowther B, 1994, Zhao F., Lowther B., Oman P., and Hagemeister J, 1993].

5.9.1 Measuring maintainability in XXS,XS,S

The maintenance of the software is important because it takes at least half of the production cost. Therefore large companies have the maintenance group to handle the things with maintenance of the software. The reasons, which make maintainability difficult are: poor quality of older existing software, systems not designed with maintenance in mind, maintenance does not necessarily attract the

best software talent. Since there are very less number of employees in the small organisations it will be very difficult to allot some staff for maintainability. Therefore they should concentrate on the above facts and develop the product.

5.10 On Estimating a "New organisation Project"

When we consider a estimation of project of a newly formed organisation like XXS where no one in the organization have any previous experience, the first time we perform a project we deal with much more uncertainty and there is no way out of it except to proceed with caution and manage the project carefully. These projects are always high risk, and are generally under-estimated regardless of the estimation process used [Vigder, M.R. & A.W. Kark, 1994]. Knowing these two facts, we must

- make the risks very clear to management and customers,
- avoid making major commitments to fixed deadlines, and
- re-estimate as we become more familiar with the domain and as we specify the product in more detail. Selecting a project lifecycle which best accommodates the uncertainty of new-domain projects is often a key step that is missing from the development process. An iterative life cycle such as the Incremental Release Model where delivery is done in pieces, or the Spiral Model where revisiting estimates and risk assessment is done before proceeding into each new step, are often better approaches than the more traditional Waterfall Model.[Vigder, M.R. & A.W. Kark, 1994]

XS, S organisation are probably working with several products therefore they will have some historical data which ease them in prediction of the future project and furthermore there are couple of employees with various experiences. When estimating cost in the above organisations COCOMO works well because they have to perform all the functions that large organisations will perform. Like estimating effort, size, schedule and more over the number of employees are not so few as XXS. Comparing to Putnam's SLIM model to the COCOMO II. The last is better because SLIM models needs large lines of codes. Since they are small there is a probability of developing small projects.

5.11 Conclusion

These are the measures suitable for the project management in the small-scale enterprises. To have a clear idea of the chapter I have summarized them as in below table 5.5. These are the measures for cost, effort, schedule, quality, maintainability etc.

Table 5.5: Suitable Measures of measurable factors in XXS,XS,S

	XXS	XS	S
Size	LOC is suitable	LOC & Function Points are suitable	LOC & Function Points are suitable
Effort	If it is old company Analogy is suitable. COCOMO & SLIM not suitable	Analogy and COCOMO are suitable. SLIM not Suitable	Analogy and COCOMO are suitable. If large project SLIM is suitable
Schedule	Analogy is suitable if it is old company	Analogy is suitable	Analogy is suitable
Cost by Anology	Suitable if it is old company	Suitable	Suitable
Cost by COCOMO	Not suitable	Suitable	Suitable
Cost by SLIM	Not suitable	Not suitable	Suitable

Conclusion

The goal of both large and small-scale organisations is to exist in the market of software development market and to make profit. This is possible if and only if they are successful in producing software. Software production is considered successful when the product is produced in time, within budget and with good quality. For any organisation to produce a successful product it must have the efficient project management. Efficient project management is possible suitable measures are used. Project measurement helps project management by providing information with which it can track and control the sudden conflicts aroused in developing project. Hence either in a large-scale organisations or small-scale organisations project management requires software measurement for effective results. The goal of thesis was project management measures for SE's the work of this research is organised in the below manner.

1. Described the project management with respective to different issues.
2. Project management and its relation with project measurement is discussed.
3. Described the importance of project measurement and use of metric information by managers, engineers and customers is described. COCOMO and SLIM models are described.
4. Classified the small organisations into XXS, XS and S.
5. Considered the measurable factors in the development process like quality, cost, schedule, effort, maintenance etc, and analysed them with small organisation in all the three cases. i.e. with XXS,XS,S. and suitable measures are presented.

Research in project measurement is mainly done for large organisations therefore tools and models are available for large organisation. Since the attention for SE's is low we do not have specific tools and models and hence more research is needed.

References

1. ***[Ahituv, N., Zviran, M., Glezer, C, 1999]*** Top Management Toolbox for Managing Corporate IT. Communications of The ACM 42.
2. ***[Albrecht, A.J., Gaffney,1983]*** Software Function, Source Lines of Code, andDevelopment Effort Prediction: A Software Science Validation. IEEE Transactions on Software Engineering.
3. ***[Athey, 1998]*** T., Leadership Challenges For the Future. IEEE Software 15.
4. ***[Basili V, 1993]*** Applying the Goal/Question/Metric Paradigm in the Experience Factory, 10th Annual CSR (Centre for Software Reliability) Workshop, Application of Software Metrics and Quality Assurance in Industry, Amsterdam, Holland.
5. ***[Bache R & Bazzana G, 1994]*** Software Metrics for Product Assessment. London: McGraw-Hill
6. ***[Bersoff, 1997]*** Elements of Software Configuration Management. In: Dorfman, M.,Thayer, R.H. (eds), Software Engineering. Los Alamitos: IEEE Computer SocietyPress.
7. ***[Boehm, Barry W, 1981]***, Software Engineering Economics, Prentice Hall].
8. ***[Boehm, B.W., Abts, C., Clark, B., and Devnani-Chulani, 1997]*** COCOMO II Model Definition Manual. The University of Southern California.
9. ***[Boehm et al, 1997]***
10. ***[Boehm et all, 1995]*** Cost Models for Future Life Cycle Processes: COCOMO 2.0.Annals of Software Engineering 1.
11. ***[Booch G, 1996]*** Object Solutions - Managing the Object-Oriented Software Project.Reading: Addison-Wesley.
12. ***[Boegh, J., Depanfilis, S., Kitchenham, B., Pasquini, 1999]*** A Method for SoftwareQuality Planning, Control, and Evaluation. IEEE Software 16.
13. ***[Buckley, 1996]***, Implementing Configuration Management: Hardware, Software, and Firmware (Second Edition). Los Alamitos: IEEE Computer Society Press.
14. ***[Chidamber, S.R., Kemerer, C.F, 1994]*** A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering 20.
15. ***[Coleman D., Ash D., Lowther B., and Oman P, 1994]*** Using Metrics to Evaluate Software System Maintainability.
16. ***[Constantine 1993]*** Work Organization: Paradigms for Project Management andOrganization. Communications of The ACM 36.
17. ***[Davis, A.M., Hsia, P 1994]*** Giving Voice to Requirements Engineering. IEEE Software.
18. ***[Deveaux, 1994]***, Counting function points. In: Keyes, J. (ed.), and Software EngineeringProductivity Handbook. New York: McGraw-Hill.
19. ***[Douglis, Charles, 1998]*** Cost Benefit Discussion for Knowledge-Based Estimation Tools.

20. ***[Fayad, M.E., Cline, 1996]*** Managing Object-Oriented Software Development. IEEE Computer.
21. ***[Fenton, N.E. and Pfleeger, 1997]*** S.L.. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press.
22. ***[Fenton Norman, 1995]*** Software Metrics: A Rigorous Approach, originally Chapman and Hall, 1991 reprinted by International Thompson Press.
23. ***[Frank Houdek, Hubert Kempter, 1997]*** " Quality patterns --- an approach to packaging software engineering experience ".
24. ***[Gilles, Alan E. & Dennis Barney, 1995]*** Metrics Tools: Software Cost Estimation.
25. ***[Goether, Wolfhart B., Elizabeth K. Bailey, Mary B. Busby, 1992]*** Software Effort and Schedule Measurement: A framework for counting Staff-hours and reporting Schedule Information.
26. ***[Goodman P, 1993]*** Practical Implementation of Software Metrics. London: McGraw Hill.
27. ***[Grady R.B, 1992]*** Practical software metrics for project management and process improvement. New Jersey: Prentice Hall
28. ***[Haywood, 2000]*** M., Working in Virtual Teams: A Tale of Two Projects and Many Cities. IEEE IT Professional 2.
29. ***[Holtzblatt, K., Beyer, 1993]*** Making Customer-Centered Design Work for Teams.Communications of The ACM 36.
30. ***[Humpry watts, 1995]*** A discipline for software engineering, Addison Wesley.
31. ***[IEEE, 1990]*** IEEE Standard Glossary of Software Engineering Terminology, report IEEE Std.
32. ***ISO-9126*** is a Software Product Evaluation Standard published by the International Standards Organization (ISO) in 1991
33. ***[Jacobson, I., Booch, G., Rumbaugh, 1999]***, The Unified Software Development Process.Reading: Addison-Wesley.
34. ***[Jarke M, 1998]*** Requirements tracing. Communications of The ACM 41, December.
35. ***[Jones C, 1991]*** Applied Software Measurement. New York: McGraw-Hill.
36. ***[Jones, C, 1996]*** Applied Software Measurement. McGraw-Hill.
37. ***[Kulik, 2000]*** A Practical Approach to Software Metrics. IEEE IT Professional.
38. **[Keil M, Cule PE, Lyytinen K, Schmidt RC, 1998]**, A Framework for Identifying Software Project Risks. *Communications of The ACM* 41.
39. ***[Lavazza, 2000]*** Providing Automated Support for the GQM Measurement Process".IEEE Software 17.
40. ***[Lindvall, 2000]*** M., Rus, I., Process Diversity in Software Development. IEEE Software17.
41. ***[Maxwell, K.D., Forselius, 2000]*** P., Benchmarking Software Development Productivity.IEEE Software 17.
42. **[Mills, H.D., Dyer, M., Linger, R.C, 1987]** Cleanroom Software Engineering. *IEEE Software* 4.

43. ***[McConnell, Steve, 1996]*** Rapid development, Taming wild software schedules, Microsoft press.

44. **[Möller K H, Paulish D H, 1993]** Software Metrics- A Practitioner's Guide to Improved Product Deveopment. London: Chapman & Hall.
45. ***[Park, R. Software Size Measurement, 1992]*** A Framework for Counting Source Lines of Code. Software Engineering Institute Technical Report.
46. ***[Pillai, K. and Sukumaran Nair, V.S, 1997]*** A Model for Software Development Effort and Cost Estimation. IEEE Transactions on Software Engineering.
47. ***[Pressman, 1997]*** Software Engineering A Practitioner's Approach. McGraw-Hill.
48. ***[Reel, 1999]*** Critical Success Factors In Software Projects. IEEE Software 16.
49. ***[Reifer, D.J, 2000]*** Requirements Management: The Search for Nirvana. IEEE Software17.
50. ***[Rettig, M., Simons, G, 1993]*** A Project Planning and Development Process for SmallTeams. Communications of The ACM 36.
51. ***[Siddiqi, J, 1994]*** Challenging Universal Truths of Requirements Engineering. IEEESoftware 11, March 1994.
52. ***[Shaw, Garlan, D, 1996]*** Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs: Prentice Hall.
53. ***[Shoemaker, D Jovanovic, 1999]*** Engineering a Better Software Organization. AnnArbor: Quest Publishing House.
54. ***[SWEBOK, 2000].***
55. ***[Tertu orci & Laryd, 2002]*** CMM for small organisations for level 2.
56. ***[Tom Gilb, 1998]*** Principles of Software Engineering Management. Addison Wesley.
57. ***[Vigder, M.R. & A.W. Kark, 1994,]***Software Cost Estimation and Control, National Research Council Canada.
58. ***[Whitmire, 1993]*** Applying function points to object-oriented software models. In:Keyes, J. (ed.), Software Engineering Productivity Handbook. New York: McGraw-Hill.
59. ***[Whittaker, 2000]*** What Is Software Testing? And Why Is It So Hard? IEEE Software 17.
60. ***[Wood ward S, 1999]*** Evolutionary Project Management. IEEE Computer 32.
61. **[Wayne Sherer S, Kouchakdjian A, Arnold, P G, 1996]** Experience Using Cleanroom Software Engineering. *IEEE Software* 13.
62. ***[Zhao F., Lowther B., Oman P., and Hagemeister, 1993]***,"Constructing and testing software maintainability assessment models", In Proceedings of First International Software Metrics Symposium, IEEE, 1993.
63. **[Zuse H, 1991]** Software Complexity-measures and methods. Berlin: Walter de Gruyter& Company