INTRODUCTION TO SOFTWARE QUALITY

1

WORLDWIDE, SOFTWARE DEVELOPMENT ORGANIZATIONS are becoming much more concerned with the process of developing quality software. Many software organizations have already established specialized groups to assess and define formal processes for development. Not all authorities agree that formal development processes are of overriding importance. Certainly, quality systems certification is becoming more important. CMM, ISO 9000 (its European equivalent, EN29000 and the various national versions) are turning into strategic instruments for many organizations. As public procurement authorities are basing their purchasing decisions on such certification, it has even become a matter of corporate survival. The growth of certification importance is due to the software project world becoming more competitive and precarious. Customers and suppliers are both seeking to shift the burden of risk-taking onto others. On the other hand, risk-taking is potentially very profitable. In order to limit the risks, project management must adopt new proactive professional approaches.

This book presents a methodology that controls risk using quality management integrated with advanced software project management. The methodology is practical and implementable so that you can use it *now*. Furthermore, adopting quality management procedures will prove valuable no matter whether your intention is simply to improve your software production or to achieve certification in some standard such as ISO 9000 or CMM or IEEE1074. Note that considerable attention is placed upon continuing to improve development and product practices. For those who are aiming at certification, remember, once ISO 9000 (or CMM, or whatever) certification is won, the job is not

over. Certification must be maintained and renewed, usually on a yearly basis.

The emphasis of the 1990s is global competition. This has caused a tremendous increase in the awareness of quality as a prime strategic weapon. The unprecedented speed at which the ISO 9000 series of standards have become the *status quo ante* of quality systems is the best possible proof of this statement. Even if this standard is "replaced" by something improved, this does not change the result, only strengthens it. Quality systems certification is a ripening concept in the industrial world. Anyone not doing it is going to be left behind.

Your next question is probably: " ... but this book is about software ... ?" That is part of the point. Do you really believe that if purchasers (customers, those pests that just happen to pay the bills) who are increasingly accustomed to demanding and receiving quality are going to continue to make an exception for software? Do you really think that we can continue to say silly things, as we all do in our "warranty" statements, that we do not accept any responsibility for our systems? How long do you think the courts of the world are going to continue to accept that?

Many countries have now established ISO 9000, CMM level 2 or 3, or an equivalent, as a prerequisite for purchases by government authorities. Since governments always buy a lot, this is going to help these countries to compete against you. This is not something that is going to happen five years from now. This is happening *now*, in the software that you are competing against.

> The ISO 9000 series has quickly been adopted by many nations and regional bodies and is rapidly supplanting prior national and industry standards. It has been adopted by the European Committee for Standardization (CEN). One of CEN's purposes is to harmonize quality standards and eliminate trade restrictions within the EEC.
>
> (*Quality Progress, May 1991*)

The objectives of this book are to guide the planning and organizing for quality in the software produced by organizations. Among the procedures to be implemented are the processes of organizing various methodologies for quality metric analysis. In particular, instructions on how and when to deploy them.

Total quality management (TQM) means the use of control techniques for making and achieving goals. These goals are usually taken to mean all of the company's goals. Somehow, software has usually managed to remain the exception. Even in very well-managed organizations, software frequently runs out of control. Why? Mostly because we simply do not really fully understand how to control the creative processes. Software TQM must include the use of plans, analysis and control

of software and the goals that cause (or allow) quality software "to happen." TQM for software includes extensive and detailed explanations of software quality assurance plans and the processes of planning. The standard for plans to be used is based on a mixture of the IEEE standard for software quality assurance plans as well as several other US and international standards.

Quality, as a Management Information System

The purpose of a management information system (MIS) is to supply *management* with *information* required to reach policy decisions, make plans, set objectives and exercise control over operations and to ensure that those objectives are achieved. However, an information system is unlikely to prove successful unless combined with sound *strategic planning* – short-term goals must be based on long-term plans. Without strategic planning, management becomes opportunism with limited chances for success. It is important that such a management information system be as simple as possible to use, and produce only useful information. In common with all such systems, it is very dependent upon the quality of its *input data*. For the past 150 years, systems analysts have labored to improve management, manufacturing and sales/distribution processes. While in many cases, the best information systems may very well be manual, for the past 35 years, this has increasingly been performed by computer (or rather, with the aid of one). It has long been axiomatic that complex activities can only be managed via *accurate* information systems.

It is known that sound *budget* and *costing systems* are vital as sources for planning and pricing decisions. A basic control procedure consists of a comparison of actual costs with expected costs and of actual volumes with standard volumes. When variances begin to infer an increase in risk, the information system must provide for communication to management so that remedial action may begin. Other commonly expected reports to management may be weekly production or backlog reports. Information should include rudimentary production control with links to *inventory* and *scheduling*.

This type of reporting takes place for almost all endeavors today – except that of creating and maintaining software. It is our belief that this has been the case primarily because the process has been basically little understood, or perhaps even misunderstood. Very recently, great progress has been made in the understanding of the software process. This includes coalescing and maintaining definitions of processes. Among the functions supported partially or fully by software engineering methodologies, are the following (see Figure 1.1).

- Definition of software development processes and life-cycle models.
- Tailoring of organizational standard processes to a defined project-specific development process.
- Production of plans, including tasks, resources and technologies.
- Guided enactment of process tasks.
- Tracking of project progress and measurement of work products and process performance (including modifications, as needed).
- Reviews and audits of defined project artifacts for conformance with internal and external standards.
- Selection and use of pre-defined generic and standard software engineering artifacts, if desired, based on a variety of de-facto and official international standards.

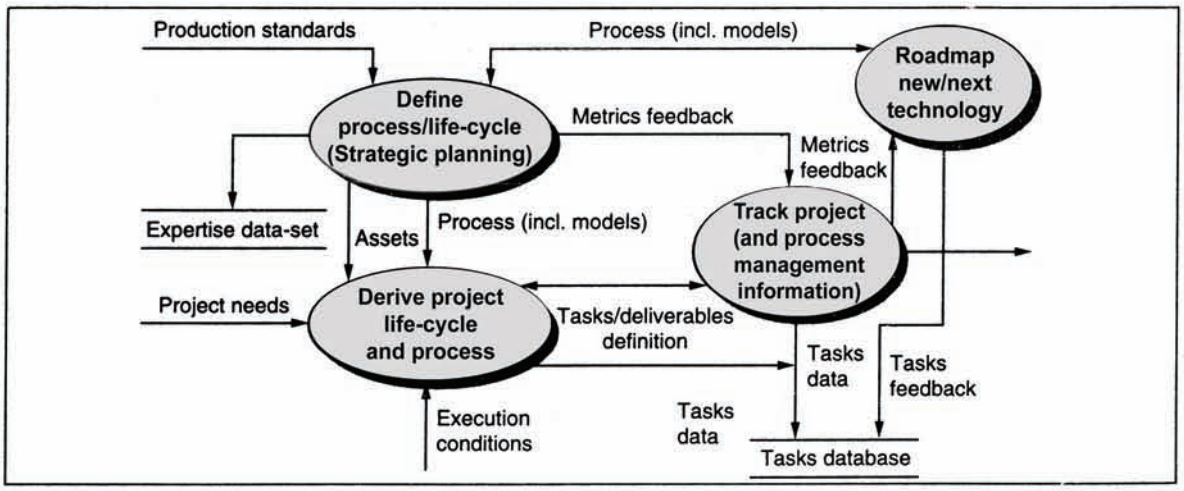

Figure 1.1 Software development processes, as a system.

The underlying philosophy of the system/methodology is the following:

- We believe that the process of creating and maintaining software is now sufficiently understood that an **information system** for the *management of information systems*, can be implemented. This is discussed in detail in the section on records collection and quality metrics.
- We believe that a system which takes into account only processes or only deliverables, is basically insufficient. At the very minimum, both *must* be accounted for. Preferably, more than that. The checklists accompanying this book show, extensively, how this is done.

- We believe that a primary difficulty facing any systems analysis is to ascertain the existence of a continuous source of accurate and dependable data. The task of gathering this data must be performed without becoming a burden.
- We believe that software quality assurance (coupled with software engineering) is systems analysis of the software development process.

At some philosophical level, it could be said that this book is about: **systems analysis, of systems analysis**. Or more properly, a systems analysis view of how software development and maintenance needs to be better managed.

Software Modeling and Commonly Used Models

There are many models in use today to describe what software professionals do and how software is, or may be, produced. Some of these models are very good, some acceptable and others rather less than adequate. Some of these are very generic and some are very parochial. In developing the approach for this book, there were a number of choices as to which models to use and which would be largely ignored. In the end what governed the choice primarily was: (a) that the model addresses efficiently the subject chosen, i.e., software quality management and (b) those models which are most important in the marketplace. If you will, those that have the best "salesmen" – though this does a certain disservice.

For example, there are the so-called "waterfall model" and "spiral model" of approaching software development management. These are excellent, for what they cover. However, they only cover very little.

Another example, is a model called "Trillium"[1] developed by Bell Canada, Northern Telecom and Bell-Northern Research. This is an excellent model and deserves more attention than the industry has awarded it. Unfortunately, it is not very well known, possibly because it was developed to be specific for a particular environment. As its name implies, it is designed for the Telecom market and hence is too narrow in scope to be widely adopted, or for this book.

There is also the IEEE Std 1074 model for software life-cycle management.[2] This is both reasonably good and includes thorough coverage (though it is overly complex). Perhaps this is the reason that it does not seem to be used very much – only by a few organizations inside the USA and almost not at all outside. Most of this book is devoted to the view of software quality management as defined by the model in IEEE Std 730.

In the end the choice was to base the book upon the IEEE model for software quality planning, and to compare it with the ISO 9000 model for quality management, even though this is very far from software, and the Software Engineering Institute's (SEI) Capability Maturity Model (CMM). The reasons for the ISO 9000 model is first in the communality in the titles and second (admittedly) its importance in the marketplace (popularity). It is very important for this book to be useful and applicable. Much more important then for it to be "academically correct" whatever that means. The reason for choosing the second is slightly more complex. Certainly, it should be chosen for its technical merits. But it covers all of the software life-cycle, which is rather more than needed for this book – software quality management is a "nonphase oriented" task. The details of the software life-cycle are not directly of interest. However, the CMM model is too important to be ignored in this book. In the end, the decision there was mostly a question of what would be most relevant to readers.

One final word about models. Throughout the book a lot will be written about all three models (ISO 9000, CMM and IEEE Std 730). It is important for the reader to understand that this book is attempting to discuss the best possible ways for assuring the quality of software that is being developed or maintained. The standard models have different objectives. ISO 9000 discusses organizing a **quality management** function for a company. CMM discusses how to assess the **processes** being used for developing software functionality. These are different. However, the *model at the head* of this effort is IEEE Std 730. Its objectives are precisely the same as those of this book (other than the question of criticality of the software, which is avoided). We may make critical remarks about ISO 9000. However, these critiques are only in terms of the goal that this book is discussing. Please, do not misunderstand our intentions. We think that ISO 9000 is an excellent standard, but its goals are not the book's goals. We may make critical remarks about CMM. Once again, these critiques are only in the terms of our goals. CMM is an excellent tool if process assessment is your goal. It is not the goal of this book. That is why they are both secondary while IEEE Std 730 is the primary model. One can compare models that have affinity, without claiming that they are the same and without forcing the issues.

One of the most basic, and important tasks of quality assurance is evaluation; that is to know the value of what you are doing and the value of any possible improvements, whether this is self-evaluation, self-assessment or an external assessment. This is one of the basic goals of this book. To help you to understand how to go about doing this for yourself and your company, and to do this quickly and inexpensively. As external assessments cost in the order of tens of thousands of dollars, this alone may be all you need the book for.

Webster's dictionary[3] defines **evaluate** as:
'To determine value of; appraise; express numerically.'
While the same book defines **assessment** as:
'Act of appraising and fixing proportionate levy; the amount at which a property is valued for taxation.'

While the people performing *assessments* of companies are not referring to taxes, the concepts of appraisal and proportionment are appropriate.

The Structure of the Model

This book intends to serve the software developing organization as a tool for designing and implementing **software quality assurance and software quality management** functions. In this sense, the book is eminently suitable for use with ISO 9000 or the SEI's CMM. (More on both of these later. If you do not recognize these acronyms, do not worry about it, you will. Otherwise you would not be reading this book.) While this is not a "cookbook," it is intended to allow fast and convenient implementation of quality management of the software process. As such, the book includes specific subsections to help the practitioner expedite task implementation.

Part of any quality implementation process must include tools which can be used to readily evaluate the quality level of a task or process being performed. For this purpose, we provide various checklists. Some of them are for management of the software process, as performed by the installation (or project). In this case, they will be similar to the kinds of things you will see when you are externally audited for ISO 9000 certification or for CMM "level *n*" certification. But at least as important, you will also use them for both self-evaluation and for evaluating any subcontractors. (If you try to use them to evaluate an employee you will be hung up to dry over a burning briar patch! This is not what they are intended for.) In any case, the major problem addressed is a swift and accurate collection of data that reflects how processes are being performed. In most cases, there *is* a process of some kind for management of the software process. What we are adding, is the ability to evaluate *quickly* and *consistently* both of which are very important. In addition to the checklists, there are also worksheets and forms which have proven very useful and applicable.

Guidebook Design Concepts

This book is designed to benefit all those organizations which are too large to be managed trivially – that means more than, say, five people. There is a basic need to simply get started! This is, by no means, an easy thing to do. Many projects, staffed by first-rate professionals, simply have no idea how to begin the subject of quality management. This puts them off the subject and so they do nothing. At very least, this makes their competition very happy. The project team may think that they are working "lean and mean." Usually they are just working much harder than is necessary. For some reason, quality management of software is still not taught, certainly not as part of the computer science curriculum. Viewing the development of the computer milieu, it is obvious that an increased emphasis upon optimization of management techniques could potentially benefit the majority of projects.

In utilizing the tools of quality management, we can more easily know what we are supposed to build, what we are building and what we have built. Not only that, but we can measure progress. That does not mean we cannot use quality management incorrectly. (Remember the old rule of the system's analyst: "computerizing a sloppy factory, makes for computerized slop!") With almost no effort, anything can be made to go wrong. However, the tools of quality management, when implemented intelligently and with proper forethought, can serve as the focal point for a real discipline, at a very low cost to the project. This means that not only can we know what has happened, we can support the product effectively. We can repeat the good things we have done and learn from them.

The general structure of the book is intended to imitate, as much as possible, the structure of IEEE 730. This is an excellent model for the design of the quality assurance function – the function and the plan of operations for the function will then appear similar. Unfortunately, there are still some weaknesses in the standard. These are discussed later on in the book. Where the standard is lacking, we have taken the liberty to expand upon the subject. Occasionally, the reason for expansion is not really a weakness in the standard or a lack of some function, but a decision by the various committees upon a specific delegation of tasks to different documents. The best example of this is the ideas of reviews and audits. The standard defines them in a very minimal manner because they are relegated to a different standard (IEEE 1061, Verification and Validation). This is, of course, perfectly legitimate and proper. Delegation of authority is a necessity of any management. So it is, and was for the committees that wrote the standards. However, this book needs to address all relevant issues so that you, the practitioner, can have the needed tools to accomplish your job, quickly and efficiently.

The Three CPIs

The concept of TQM is beginning to spread throughout the world and is now starting to make an impact on software development organizations. This is more then just a "new fad." The implementation of TQM in an organization *forces* management to push the primary focus of all activities in a direction of continuous process improvement (CPI). Continuously improving all of the organization's processes is, in any case, very difficult. In the "western world" (Europe and North America) productivity enhancements over the past 100 years have averaged about 4.5% per annum for industrial processes and about 0.9% per annum for services. Clearly, the competitive imperative forces us to at least understand this, but even more so, to improve upon it. The concept of TQM for software is an overall technique for organizing and using methods of quality planning and metric analysis, such that results obtained are usable and *repeatable*.

The concept of *software total quality management* is intended to guide the quality assurance practitioner in planning and organizing for quality of software produced by the organization from which his or her living is coming. Quality planning is a primary tool – you cannot know where you are going if you do not know where you intended to go. Clearly, a first real step must be building the quality assurance (QA) plan and managing the quality goals defined by the plan. The quality task is particularly difficult when attempting to plan the quality of software, which is only little understood. This is true even for engineers who have had quality assurance training (whether hardware engineers or software engineers). Such training itself is rare enough. For those engineers (even first-rate professionals) who have not had such (QA) training, very little understanding exists of the issues involved *vis-à-vis* software.

Experience has repeatedly demonstrated that attempting to understand software quality and the assurance of software's quality, as simply an extension of software engineering techniques, is doomed to fail. **Software engineering is an iterative system for knowledge acquisition**. The major problems in systems today are error fixing and the problems caused by this (collectively called maintenance). These problems are most poignantly described by error propagation, which is the logical equivalent of wear-down in hardware. "Total quality management for software" acts firstly as an aid to the process of organizing the various methods for quality planning and metric analysis. But also, to instruct the user in proper methods of employing these methods, such that results obtained are usable and repeatable. Too often, the literature is rife with "statistics" which cannot be independently verified.

Quality metric analysis is used as a tool for understanding effects of various variables upon the way systems are developed and maintained. This must have both an immediate and a long-term effect upon systems acquisition. The knowledge thus gained is incorporated into the QA plan (of the system, and/or the software). It then becomes a primary management aid for enhanced understanding and improved management of the systems: in operation, in production and in acquisition by the organization. This feedback is then used by management for bettering the quality goals defined via the QA plan. No feedback mechanism can be of value if these input values cannot be verified.

While planning for quality, as in any other management discipline, a good understanding of "where you are going" – what your quality *goals* are – is a prerequisite for effectiveness of the planning process. Knowing "where you are going" means to be able to plan what the quality attributes are to be, and what the proposed "relative value" (quantitative) of each attribute *needs* to be. Not only that, but all this usually needs to be done before the system itself is particularly clear. The software QA practitioner needs to have a set of tools for both professional purposes, and for allowing project management/client management to quantify **their** quality goals for the project/product in development. (To eliminate overuse of the "/" double terms, for the remainder of this book, the words "project" and "product," and the terms "client management" and "project management" will be used interchangeably.)

Most organizations selling consulting for TQM claim, at least in their marketing literature, that the type of organization is not relevant to their techniques. This is quite simply not correct. When one of the authors discussed the questions with some of them (and actually discussed it personally with executive officers of Crosby Associates International Inc., Conway Quality Inc., Organizational Dynamics Inc., and representatives of the Juran organization) they all acknowledged that they had no real understanding of software and had never analyzed when software was similar to other kinds of problems and when it was not. In instances of implementing their techniques there have been very pointed differences, where a software-oriented organization has been concerned. Particularly, when I mentioned to them that in software there is no manufacturing process, but only design, they were universally unhappy. This forced them to re-evaluate the comment that organization type is irrelevant. The result is that they may need some sort of a "mental fix" – actually, we all do. What is more important for us, is that this now causes us to change the mode of thought based upon the software producers being accustomed to being always "feared" by the organization. Nonetheless, they still think that way, and it is still always the case that we cannot afford the luxury.

TQM uses control techniques for making and achieving goals. This is usually taken to mean all corporate goals. Somehow, software has usually managed to remain the exception. Even in very well-managed

organizations, the software frequently runs out of control. Why? Mostly because we simply do not really understand how to fully control the creative processes. Software TQM must include the use of plans, analysis and control of software and goals which allow for quality software to happen. Quality planning is a primary tool. Clearly, a first real step must be establishing an agreed baseline, followed by building a quality plan and managing defined goals. A basis of control must be an understanding of how software quality may be measured. It is imperative that we base analysis upon these measurements. This analysis must have both an immediate and a long-term effect upon systems acquisition. This ensures deeper understanding and allows better management of the system: in operation, production and in acquisition by the organization.

Total Quality Management Practice

As stated above, the usually accepted concepts of TQM, as promulgated by the various "management gurus," are based upon the same concepts being discussed here. The only difference is that your normal, everyday management guru does not know software. They seem to think that products just happen. Clearly, this is not the case in a software company. Nor is it the case in an engineering company that develops and markets products which happen to be software rich. Many products today may have as much as a 90% software content (measured in terms of total investment in their development). In discussions that one of the authors has had with several of these highly skilled professionals (and, do not misunderstand us, we have a great deal of respect for these people and both admire and learn from their work), they have always emphasized that they really did not understand software, but they thought it was essentially the same as anything else! This is not quite accurate. Figure 1.2 on the next page, shows a *typical* enterprise structure. Notice that software is simply not there. Why? Is this simply an oversight on their part? Hardly! These people are much too professional and experienced for that. We must be aware that certain kinds of systems are too complex for trivial analysis (e.g., traffic, weather, biological, economic). Major information systems of all kinds, whether oriented towards information processing or real-time may be as complex as biological systems. In any case, the discussion centers around the idea that one must *plan* to do better and that all these ideas are a challenge to traditional management philosophy.

The difference then, is that software is all development (design). Remember, industrial processes are intrinsically stable – they are algorithmic processes. Information sciences are intrinsically unstable. The usual models just do not work.

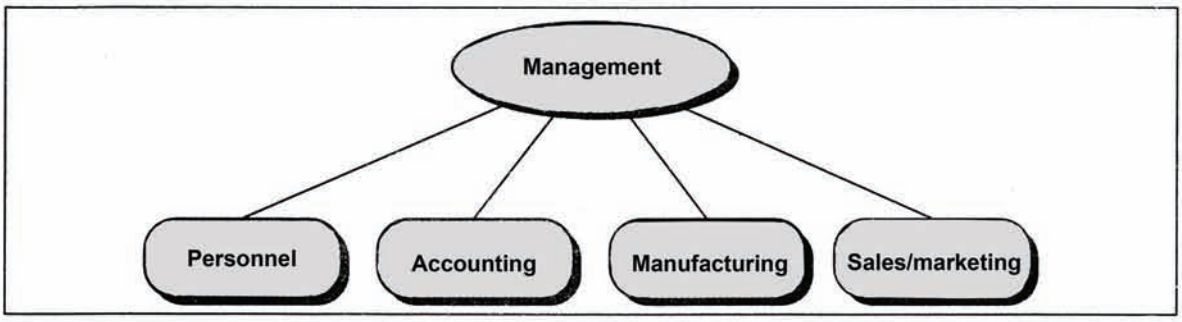

Figure 1.2 GURUware.

Well, if the usual models do not work, what, if anything, can be learned from them?

The answer is: the three CPIs!

- Continuous process improvement.
- Continuous product improvement.
- Continuous productivity improvement.

Interesting, is it not? We said three different things with the same acronym. The great big secret is, in software they are all the same. The stupid expression; "If it ain't broke, don't fix it!" is simply that, *stupid!* That is the surest way to not stay in business for any length of time. Even monopolies understand that today.

We are asking you, on a continuous basis, to think about what you are doing, about how you are doing it and about how much it is (and should be) costing you. We are asking you, on a continuous basis:

- to improve your *products*
- to improve your *processes* and
- to improve your *productivity*

As TJ Watson said, we are asking you to *think*. Nasty, isn't it?

The First Steps to Planning for Quality

An organization engaged in the development, acquisition or maintenance of software – whether for profit or as an internal service to the company – must have a view of quality and its meaning to them in relation to the proposed product. Our wish at this point is to suggest what that view might be to achieve the sort of results we think you desire. Look at Figure 1.3. It displays the relationship with which management needs to be

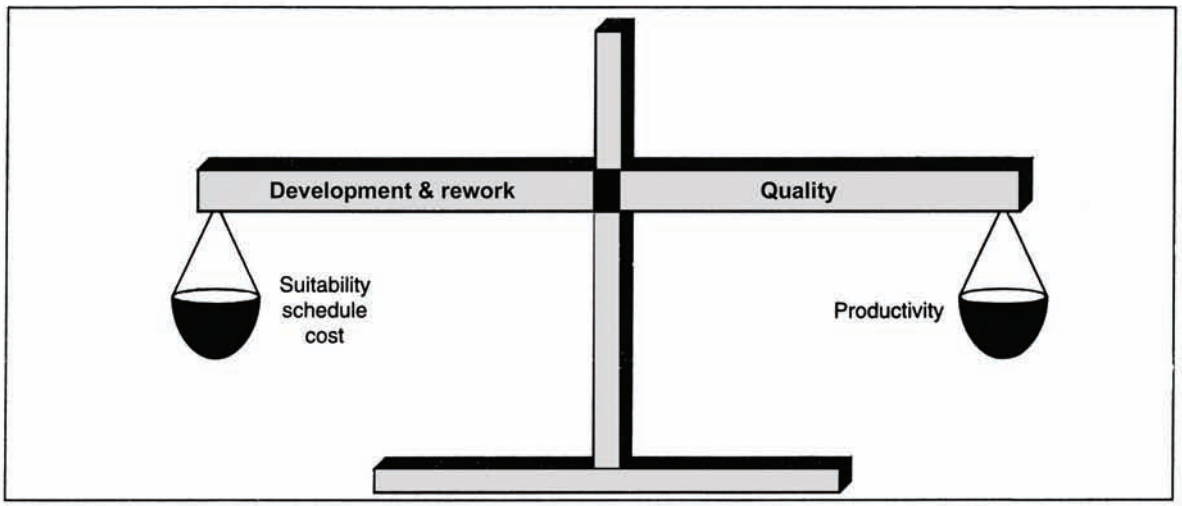

Figure 1.3 Management relations for quality.[6]

concerned for software. Let us use this drawing to establish a few basic principles and put them in their place. Let's look at the right-hand side of the drawing. Notice that quality is the *arm*, from which *productivity swings*. **In software, quality and productivity just do not separate!** They are part of the same thing. This is axiomatic to any understanding of quality, as it relates to software. Crosby's famous book said "Quality is free." This is not quite true (close, but not quite). What is absolutely true (certainly for software) is that what really is costly, is a lack of quality. Generating, then hunting and fixing bugs is a very expensive, wasteful and unproductive way of life.

There is a major difference between software and hardware. **In software the defects are built in. In hardware they are a function of time**. Understanding this is a prerequisite to a plan for quality.

Now let's look at the left-hand side. Development *and* rework are the balance arm of quality. What swings from them are the three keys to any project's success: *suitability* of the produced system to the clients' expectations, *delivery* on the agreed date and *price* that is considered appropriate. Now isn't that a nasty lot of things to say? If you do not like it, you deserve it.

To be kept in perspective, quality concepts must be understood in terms of accepted definitions. "Accepted" means definitions as used by commonly used industry standards. The standards "ANSI/IEEE STD 730" and "ANSI/IEEE STD 610.12"; the American National Standard Institute/Institute of Electrical and Electronic Engineers Standards for Software Quality Assurance Plans and for Software Engineering Terminology. From those sources, one can see that the definition, is actually composed of two parts. The definition of what quality means:

> The totality of features and characteristics of a product or service that bear on its ability to satisfy given needs (ANSI/ASQC A3-1978).

And, the definition of what quality assurance means:

> A planned and systematic pattern of all actions necessary to provide adequate confidence that material, data, supplies, and services conform to established technical requirements and achieve satisfactory performance.

The software QA plan provides the necessary framework for planning of:

> ... systematic actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.

This sounds rather trivial. However, experience has shown that this is quite a complicated and delicate – not to mention, diplomatic – task. The total quantity of activities and tasks which need to be addressed is quite large. In a general term, we call it software auditing.

The software audit process improves the *availability and reliability* of software and the products supported by software. The process is designed to be analogous to the quality control function in manufacturing of "regular" products. This concept of quality auditing, like any quality audit concept, must be soundly based. Standards are one of the necessary parts of the baseline against which things can be measured. For our application, we are in need of more than this. We need an overall concept of planning. The resulting plan must include both the procedures and planned analysis of productivity and quality.[5]

The procedure

There are certain fundamental requirements for any quality improvement process to be successful. These requirements can be stated as a list of points as shown in Table 1.1.

Now, how is this to be accomplished? A set of four activities must be implemented by the organization.

- **First,** establish a *baseline* for measurement. Whether this process is called software practices assessment or software maturity index (as the SEI calls it) or software process model is not important. This process measures the state-of-the-practice used by the organization in its development activities.
- **Second,** develop a comprehensive quality plan, which includes productivity and goals, along with the technology goals to be used by the organization. Software engineering is immature on principles. That is why pure SE activities have not really had a major impact on the productivity of our systems (by the by, the most profound impact has come from reusable packages such as trans-

Table 1.1

Accept the quality process
Management commits to the improvement process as corporate culture
There is always room to improve what is being done
Preventing problems is smarter than reacting to them
Management focus, leadership and participation
A performance standard of zero defects
Participation by all employees, as individuals and as groups
Focus improvement on processes, not people

action monitors and databases). This plan is wholly dependent upon the total *commitment* of management. TQM, particularly for software, begins with the most senior management of the concern (e.g., the CEO).

- **Third**, implement your *plan*. The implementation must include all three types of goals (productivity, quality and technology).
- Finally, the **fourth** aspect. Everything must include measurement, measurement and *measurement*! Of course, as stated in several other places in this book, this measurement is not a religious rite. It is used, after rigorous analysis, as feedback for the software organization.

The most difficult comprehension problem is establishing your corporate baseline. Even all the corporate awareness discussed previously is not sufficient if there is not a reliable picture of where the organization stands in comparison to the industry, and what the plan for improvement needs to be. Also, of course, we cannot prove improvement unless we have means for measuring it.This implies that we need to start measuring our current state before starting to improve it.

Only after this measurement can the quality attributes be determined, along with their constituent parts. Once these parts exist and are allocated to attributes of suitability, maintainability or both, the last and final aspect may be determined, that of the quality attribute relationships. This closes the circle. Once this framework exists, determining the software quality through the product life-cycle (concept, requirements, design, implementation, testing, installation and checkout, operations and maintenance[6] and finally retirement[7]).

An excellent example of how things should be accomplished can be taken from the quest for quality at the Philips Corporation. They call this quest, "company-wide quality improvement principles." This particular example is taken from their Singapore subsidiary, and was reported in a newsletter of the American Society for Quality Control (ASQC).

- **Customer satisfaction.** A perfect interface must be achieved between company performance and customer needs in all aspects that customers consider to be important.
- **Leadership.** Quality improvement is primarily a task and responsibility of management as a whole.
- **Total involvement.** There must be total involvement of all employees at all levels and in all functions. Equally important is the complete involvement of all suppliers of goods and services.
- **Integrated approach.** Integration must be achieved between functions and between levels. Traditional organizational barriers must be removed.
- **Systematic approach.** A systematic approach must start with a clearly defined business strategy, which is then translated into an improvement policy with objectives and priorities. These must be followed by detailed planning, implementation and monitoring of progress.
- **Defect prevention.** Defects must be prevented from occurring. Performance must be the result of built-in capabilities.
- **Continuous improvement.** The approach should not have the character of a campaign or a project. Excellence can only be achieved by continuously investing in improvement, step-by-step, year after year.
- **Maximum quality.** Long-term objectives must be set which reflect the will to strive for excellence. The path towards excellence must be marked by challenging but achievable and acceptable targets.
- **Education and training.** Widespread attention will be given to education and training. A new work culture can only be realized if people are more than ever prepared to make their contribution.

The point referring to continuous improvement is particularly interesting. It "just happens" to be a fact of business life that nearly every major successful corporation in the world places continuous quality improvement as a primary corporate policy. A sadder example was reported by *Business Week* on July 4, 1988; under the title; "Missed deadlines at Lotus Development Corporation" (see Table 1.2, below). Lotus was clearly one of the brightest and most interesting of companies in the software marketplace. They enjoyed some amazing successes. They also had some very interesting failures (we can, of course, say the same for any other major software company) which have been publicly reported. An intelligent manager learns from the mistakes of others. Lotus learned their lessons, and they should be applauded for that – but it was too late, they are no longer an independent company and their upper management has been replaced. That is what the *Business Week* article was about and, in a very real sense, that is what "total quality management for software" is all about.

Problems of products being delayed, or even canceled, have multidimensional costs. These costs include both *lost product sales* and

Table 1.2 An example of what is happening in industry

Product	Announced	Promised for	Status
Modern Jazz	March, 1987	1Q 1988	Canceled June 16, 1988
1-2-3 rel. 3	April, 1987[3]	1Q 1988	4Q 1988
1-2-3G	April, 1987	4Q 1988	1Q 1989
1-2-3M	April, 1987	1Q 1988	1Q 1989
DBMS	April, 1987	4Q 1988	2Q 1989
1-2-3 MAC	October, 1987	3Q 1988	1Q 1989
AGENDA	November, 1987	2Q 1988	3Q 1988

decreased customer confidence. Both of these are intangibles which are difficult to measure directly, but this difficulty must not be allowed as an excuse to ignore them. We all remember that Lotus started at about the same time as Microsoft and in the beginning they were in competition as to who would be the leader. Lotus has now been purchased by the largest fish. They are not going to be leading anything now.

A Case Study

John Cullyer from the Royal Signals and Radar Establishment (RSRE), UK Ministry of Defence (MOD). VIPER microprocessor, a 32-bit RISC chip designed for safety-critical applications.

RSRE performed a study of NATO software in the 1980s, using a static analysis technique in which a program is represented as a directed graph, various expressions are associated with the arcs and conclusions regarding correctness are derived from them. Of the modules (not necessarily whole programs) which RSRE sampled from the NATO inventory, one in ten were found to contain errors, and of those, one in 20 (or one in 200 overall) had errors serious enough to result in loss of the vehicle or plant! About the same findings were made whether the code came from Britain, the USA, or West Germany.

VIPER is an attempt to address this problem. The project was felt to be so urgent that it was funded within 48 hours of submission. There is no stack ("We don't like stacks – they overflow"). There are no interrupts; all device handling must be done by polling. Cullyer said that it is not possible to verify programs that permit interrupts. "I don't think we have all persuaded our bosses that there is a problem. If we do not implement these methods, there will be a lot of accidents and a lot of people will die. If we do implement them there will still be accidents, but we will limit the casualties." He also mentioned that new MOD software procurement standards require formal development techniques for critical software. MOD regulations explicitly prohibit any cost

saving that might increase hazard to life – you are not allowed to trade lives off against money.

The shuttle is a totally "fly-by-wire" craft. The onboard flight control software is built from about 500,000 lines of HAL/S source code; the total error rate for the software was 0.11 errors/thousand source lines (or about 55 errors). IBM got the error rate down to 2.2 errors per 1000 lines in 1932 and to 0.11 per 1000 lines at the end of 1985. The figures referred to errors discovered by the customer and undiscovered errors remain. For instance, how they could be sure the abort sequence software was error free, since it had never actually been used? The answer is that they exercised it in a software simulator.

How can one determine how many consecutive failure-free tests would be needed to establish with 99% confidence that the probability of failure is less than one in a billion? Commonsense suggests that it must be at least a billion, perhaps more. Miller derived that you actually need around 4.61 billion, and presented the rule of thumb that to obtain confidence that the probability of failure is less than 10^{-N}, you need about $10^{+(N+0.5)}$ trials. He pointed out that in most cases it is only practical to test up to around 10^5 trials, which can only reveal bugs that appear with frequency $10^{-4.5}$ or greater.

People sometimes say that good engineering practices ensure that the probability of failure is much less than $10^{-4.5}$. This is rather illogical if testing reveals any errors at all. If the tests reveal frequent bugs, why should you believe that your good engineering practices have prevented the subtle ones? Performing binary patches to weapons software in the field is a common practice in the US services; the UK standards prohibit this. Cullyer mentioned that the UK has only 11 scientists and engineers who review the entire avionics; he estimated that there were only about 50 people in whole western world looking after civil avionics safety.[8]

Notes

1 The copy I am referring to is: "Trillium: Model for Telecom Product Development and Support Process Capability," Bell Canada, Release 3.0, December 1994. The word "Trillium" is not listed as trade marked by this document.

2 Currently undergoing a process of revision.

3 Webster's *New Illustrated Dictionary*, Editors-in-chief Allan S. Kullen and Frederick Reinstein, Books, Inc., 1970.

4 In software, quality and productivity just do not separate!

5 In 1985, software costs for the United States Department of Defense were roughly $11 billion. In the USA, as a whole, the total outlay for software was about $70 billion; worldwide the costs for software reached about $140 billion – that was 11 years ago. The present rate of growth in the worldwide software market is about 12%. (As reported by Barry W. Boehm, TRW, September, 1987, *IEEE Computer*.)

6 Operations and maintenance is called here, one stage. This is where the majority of resources are consumed. While it is true that, conceptually, every maintenance

project is a development project in the small, it is also very important to remember the additional onus that is placed upon these activities.

7 The actual fact of retirement seldom needs to be examined. By this point it is usually too late to do anything significant or to learn a great deal about it. Most software systems live for 15–20 years. At that point, no one cares anymore, they simply wish to get the old stuff out of the way and get on with the new system.

8 Taken from "Testing for the individual programmer," JC Cherniavsky and WR Adrion, NTIS, US Department of Commerce, PB 166960.