The Software Quality Economics Model for Software Project Optimization

LJUBOMIR LAZIĆ, AMEL KOLAŠINAC, DŽENAN AVDIĆ
Technical Faculty
University of Novi Pazar
Vuka Karadžića bb, 36 300 Novi Pazar
SERBIA
llazic@np.ac.yu, akolisinac@np.ac.yu, dzavdic@np.ac.yu. http://www.np.ac.yu

Abstract: - There are many definitions of quality being given by experts that explains quality for manufacturing industry but still unable to define it with absolute clarity for software engineering. To enable software designers to achieve a higher quality for their design, a better insight into quality predictions for their design choices is given. In this paper we propose a model which traces design decisions and the possible alternatives. With this model it is possible to minimize the cost of switching between design alternatives, when the current choice cannot fulfill the quality constraints. With this model we do not aim to automate the software design process or the identification of design alternatives. Much rather we aim to define a method with which it is possible to assist the software engineer in evaluating design alternatives and adjusting design decisions in a systematic manner. As of today there is very little knowledge is available about the economics of software quality. The costs incurred and benefits of implementing different quality practices over the software development life cycle are not well understood. There are some prepositions, which are not being tested comprehensively, but some useful Economic Model of Software Quality Costs (CoSQ) and data from industry are described in this article. Significant research is needed to understand the economics of implementing quality practices and its behaviour. Such research must evaluate the cost benefit trade-offs in investing in quality practices where the returns are maximized over the software development life cycle. From a developer's perspective, there are two types of benefits that can accrue from the implementation of good software quality practices and tools: money and time. A financial ROI calculation of cost savings and the schedule ROI calculation of schedule savings are given.

Key-Words: - Software Quality, Quality Cost Model, TQM, Cost optimization, ROI calculation

1 Introduction

Most important thing in analysis of the cost of quality is visibility in development cycle. It is the visibility gained form cost of quality analysis that enable the QA people involved to focus their attention on those activities which discover, and correct the root cause of the software defects. This root cause analysis (through Pareto technique or any other method) allows the QA people to determine how the development process can be improved to prevent major area cause of defects.

There are many definitions of quality being given by experts that explains quality for manufacturing industry but still unable to define it with absolute clarity for software engineering [1,12][1]. In cost quality analysis we should identify what we are trying to achieve, the goals should be defined and that should be measurable. So that analysis can verify that it is actually increasing the quality level or not.

There is very little research is available about what quality initiative should be taken and how it reduces your development cycle and improves product quality. Significant research is needed to understand the economics of implementing quality practices and its behaviour. Main focus in quality management is how to make profitable decisions on quality expenditures [2-6].

With respect to quality initiatives we can divide the organizations into two categories, one are those who do not believe that process improvement and training of the human resources would bring in any improvement in quality. They think this is extra cost. Whereas second kind of organizations who have realized the importance of processes and its continuous improvement, plus good care of the staff

[1] This work was supported in part by the Ministry of Science and Technological Development of the Republic of Serbia under Grant No. TR-13018.

and their knowledge upgrading [7-10]. Organizations those who have realized the importance of high quality and process efficiency normally find it difficult to start the improvement cycle [6-9]. They find it difficult to convince their top management to allocate budget for the quality initiatives. The real issue is of investment, not the cost. The investment in software quality, like any investment has an immediate cost, with an expected net payback. There is where Quality Cost Analysis could be used as effective tool to make them understand the ROI [7,11]. As we all know that top management does understand the language of money very well. They would like to increase sales and have more profits. Task of QA people is to relate the Cost of quality as investment and its benefits are increased revenue.

To enable software designers to achieve a higher quality for their design, a better insight into quality predictions for their design choices should be given. In this paper we propose a model which traces design decisions and the possible alternatives. With this model it is possible to minimize the cost of switching between design alternatives, when the current choice cannot fulfill the quality constraints. With this model we do not aim to automate the software design process or the identification of design alternatives. Much rather we aim to define a method with which it is possible to assist the software engineer in evaluating design alternatives and adjusting design decisions in a systematic manner.

Case studies of the success stories can be presented to the top management as tool to increase their understanding high quality and how to go about that. The major problems we see with these case studies are that there is no local research available [8,9]. The possible argument of the top management could be; that these practices do not suits our culture or our environment. What is required is the research in preparing the local case studies and research for the organizations that have implemented the TQM and those who have not. The comparison of both will provide good starting point for the management of such organization. There are some prepositions, which are not being tested comprehensively, but some useful Economic Model of Software Quality Costs and data from industry are described in this article [5-12]. Significant research is needed to understand the economics of implementing quality practices and its behaviour. Such research must evaluate the cost benefit trade-offs in investing in quality practices where the returns are maximized over the software development life cycle.

2 Software Quality Dimensions and Models

2.1 Software quality

One would expect software quality to determine the cost for developing software. It would also determine the value of the software, hence the market price. To an economist, optimal quality would normally be different from technical perfection. Before we can identify the value-maximizing strategies, we need to know more about the different dimensions of software quality and empirical evidence how quality is perceived and valued, both by the developing engineers and by users or on the market. Is it possible to come up with general models for the optimal (economical) Cost of Software Quality. The question is how important different quality dimensions are from an economical point of view: That the code is optimal? That the user-interface is perfect balance between different dimensions of software quality, e.g. between different levels of user-friendliness and long run handling efficiency?

Although no standard industry definition exists for what constitutes good quality in software, it is generally taken to mean that a software product provides value (satisfaction) to its users, makes a profit, generates few serious complaints, and contributes in some way to the goals of humanity (or at least does no harm) [1,7,11]. Software quality is difficult to define because there is no single comprehensive and complete standard definition of its lexicon. Various aspects and terms are found in sources such as ISO 9000-3, Institute of Electrical and Electronics Engineers Software Engineering Standards, and various books on the subject.

The following are the key dimensions of software quality.

• **Level of satisfaction:** The degree to which customers or users perceive that a software product meets their composite needs and expectations.

• **Product value:** The degree to which a software product has value for its various stakeholders relative to the competitive environment.

• **Key attributes ("ilities"):** The degree to which a software product possesses a combination of desired properties, e.g., reliability, portability, maintainability.

• **Defectiveness:** The degree to which a software product works incorrectly in target user environments due to debilitating operational defects.

• **Process quality:** In relation to the development

process by which the product is produced, it means good people doing the right things in an effective way.

A definition fashioned from the above aspects should be created for your organization and for each project. Every application or business domain faces a specific set of software quality issues, and software quality must be defined accordingly. For example, mission-critical applications have extremely stringent operational needs, whereas typical information system applications must focus on general measures of customer satisfaction. It also is important for each software development project to define its specific meaning of software quality during the planning phase. Such a definition contributes to the basis for setting objectives and practical measures of quality progress and determination of readiness for release to customers.

The new standard SQuaRE [13] consists of 14 documents grouped under five thematic headings:

- Quality Management, defining all common models, terms and definitions referred to by all other standards in the SQuaRE series,
- Quality Model, probably updated version of ISO/IEC 9126-1 [22],
- Quality Measures, derived from ISO/IEC 9126 and ISO/IEC 14598,
- Quality Requirements, standard for supporting the specification of quality requirements, and
- Quality Evaluation, providing requirements, recommendations and guidelines for software product evaluation.

Characteristics of software product lines as well as experience with several existing quality modeling approaches have guided us in defining three main requirements for appropriate quality modeling: *flexibility*, *reusability*, and *transparency.*

Flexibility - A quality model should be flexible because of the context dependency of software quality. There are several quality contexts: *company context*, *project context* and *process context*. *Company context* includes the unique characteristics of a specific software company where the model is used. A flexible quality modeling approach should be applicable across different companies. However, employment of the approach in different companies should result in unique quality models that reflect the unique characteristics of each single company. *Project context* combines unique characteristics of a particular software project like its domain (e.g., web application, embedded system) or different views on the quality represented by different project stakeholders.

For example, a system end-user may think about software reliability in terms of failure density, whereas a software developer may also notice the relation between reliability and software design complexity. A flexible quality modeling approach should be applicable to any project domain and incorporate views (on quality characteristics and their relationships) of all relevant project stakeholders.

Process context reflects the characteristics of a software development process like its stability or availability of measurable objects in different process phases. A flexible quality model should not assume a stable process. The modeling approach should allow creating the model tailored to company- specific characteristics of the development process. Postulating a stable process would make a modeling approach inapplicable in most software companies due to the lack of stable processes. A very important issue in quality modeling is the phases of the software lifecycle to which the model is applicable. In essence, quality modeling is more effective the earlier it can start in the software lifecycle, and the more process phases it embraces. From the perspective of controlling the quality, early quality evaluation allows timely identification and elimination of potential quality problems. For instance, elimination of a design defect during software operation could cost a hundred times as much as if the defect would be identified and removed already in the design phase.

In early phases of the software lifecycle, hardly any measurable items are available. Therefore, the flexible approach should integrate all the characteristics of a software project environment that influence the quality of a software product.

Those could be product characteristics (e.g., design complexity), process characteristics (e.g., inspection efficiency) as well as resource characteristics (e.g., designer experience). To improve model accuracy, it should also take advantage of people's experience and, besides quantitative (measurement-based) data, it should cope with qualitative input, e.g., experts' assessments. During subsequent phases of development both, the software system and the whole software project environment are the subjects of continuous change. As the project evolves, new products are developed, new processes are applied, and more measurable artifacts are available. In order

to control the quality of a software product, the quality model should evolve in parallel to software changes. The modelling approach should cope with missing data as well as allow easy re-estimation of quality evaluations, as the new and more precise data appear.

2.2 Review of Existing Quality Models

A *quality model* is the set of characteristics and the relationships between them, which provide the basis for specifying quality requirements and evaluating quality. Of course, the quality model used will depend on the kind of target product to be evaluated. In this sense, the current standards and proposals define "generic" quality models. The main problem is that these models are too general for specific areas, such as components or component-based systems design (CBSD). In this sense, some authors have started to propose particular models and metrics for software components [14,15]. In particular, our initial proposal is shown in Table 1. This model is a refinement of ISO 2196, particularizing it for components: some of the sub-characteristics disappear, and others change their meaning. See reference [14] for a detailed description of the model.

Quality Characteristic	Boehm	McCall	FURPS	ISO 9126	Dromey
Testability	X	X		X	
Correctness		X			
Efficiency	X	X	X	X	X
Understandability	X		X	X	X
Reliability	X	X	X	X	X
Flexibility		X	X		
Functionality			X	X	X
Human Engineering	X				
Integrity		X		X	
Interoperability		X		X	
Process Maturity					X
Maintainability	X	X	X	X	X
Changeability	X				
Portability	X	X		X	X
Reusability		X			X

Table1: Comparison between the quality models [2]

In addition, it is important to classify the model's quality characteristics according to several criteria [16]:

1) First, we need to discriminate between those characteristics that make sense for individual components (that we will call *local* characteristics) and those that must be evaluated at the software architecture level (*global* characteristics). For instance, Fault Tolerance is a typical quality characteristic that depends on the software architecture of the application. On the contrary, Serializable is a property applicable to individual components only.

2) The moment in which a characteristic can be observed or measured also allows establishing another classification. Thus, we have those characteristics observable at runtime (e.g. Usability) and those observable during the product life-cycle (e.g. Maintainability) [17].

3) We need to have also into account the phase within the CBSD life cycle where the quality attribute is applicable: Assessment, Integration, Testing, Operation, or Maintenance.

4) It is also important to identify the target users of the quality model, as ISO standards explicitly state. In our case, these users are mainly software architects and designers, which need to evaluate the COTS components available in software repositories (or that can be bought from software components vendors) in order to be incorporated into the software product they are building. In this sense, the model focuses more on the "programmatic" interfaces of components than on their "user" (GUI) interfaces, i.e., we are particularly concerned with the API's defining the services provided by the components so they can be composed and integrated with other programs. Other kinds of possible users include: acquirer, evaluator, developer, maintainer, supplier, end-user and quality manager.

5) For COTS components, it is essential to distinguish between internal and external metrics. Internal metrics measure the internal attributes of the product (e.g. specification or source code) during design and coding phases. They are "white-box" metrics. On the other hand, external metrics focus on the system behavior during component testing and operation, from an "outsider" point of view. External metrics are more appropriate for COTS components, due to its "black-box" nature. However, internal metrics cannot be completely discarded, since some internal attributes of a component may provide an indirect measurement of its external characteristics. These metrics are also valid to the component developer, who uses them to assess the quality of the product produces. In case of component based systems internal metrics are also needed, since they will help evaluate the quality of the composition of the internal components that make up the system. Therefore we will distinguish between four main categories: External to a

component-based system; External to a COTS component; Internal to component-based system; and, Internal to a COTS component Finally, it is important to note that there are other kind of *marketing* characteristics such as price, technical support, license conditions, etc.—not directly related to technical quality—which may be of great importance when selecting components. In this paper we will concentrate on quality characteristics only, leaving the rest of characteristics for further research.

2.3 Comparison of the Quality Models

According to [18], there are two kinds of approaches to model product quality: *fixed -model* and *define-your-own model*.

The fixed-model solution provides a fixed set of qualities so that identification of customer-specific characteristics results in a subset of those in a published fixed model. To control and measure each quality characteristic, the characteristics, measures, and relationships associated with the fixed model are used. Examples of such models are presented in articles [19-25].

They contrast the define-your-own-model approach, where not a specific set of quality characteristics is defined, but rather – in cooperation with the user – a consensus on relevant quality characteristics is identified for a particular system.

These characteristics are then decomposed (possibly guided by an existing quality model) to measurable quality characteristics and related metrics. The relationships between quality characteristics and sub-characteristics could then be defined either directly by project stakeholders (*directly-defined model*) or generated automatically (*indirectly defined model*). Directly-defined models have the form of dependency graphs and examples of such approaches are presented in: [26-30].

Indirectly-defined models result from the application of various techniques, so that software project stakeholders can influence the quality model by choosing the technique and its parameters. However, they have no direct influence on the output quality model. Quality relationships represented in such models are often so complex that project stakeholder has difficulties with understanding them. The main domains from which such models come are mathematics and artificial intelligence. There are also some known attempts to employ other approaches such as multi criteria decision aid [31]. Examples of mathematical models are: multiple regression models [32], Alberg diagrams [33], and logistic regression models [34]. Typical artificial intelligence approaches are: decision and classification trees [22], genetic algorithms [36], neural networks [37], case-based reasoning [38], data mining [39], and fuzzy expert systems [40].

Fixed -model approaches lack the flexibility requirement. They define a constant set of quality characteristics and relationships between them. However, it is unrealistic to assume that it is possible to define a prescriptive view of necessary and sufficient quality characteristics to describe quality requirements at every company, for every project and every stakeholder. There is probably some amount of quality characteristics and relationships universally true for all organizations and projects, but most of them differ from organization to organization and from project to project.

Horgan [41] tries to identify such universal characteristics to compare quality across projects. He introduces Key Quality Factors (KQFs) as common for every project and every company. However, KQFs are high -level quality characteristics like maintainability or correctness already known from ISO9126 [22] or McCall's [21] models. Such an approach limits the comparability of project quality to only high-level characteristics. Furthermore, the level of reusability of quality experience gained in past projects and stored in such universal models depends on the level of project similarity and is usually limited by the lack of indicators of similarity.

The common problem of fixed-model approaches is that they are limited to quantitative (measurement-based) and product-related data, whereas in early stages of the software lifecycle hardly any measurable products are available.

Some of the latest fixed-model methods (e.g., [25]) broaden the scope of measurement on processes and resources but are still unable to profit from qualitative data like, for example, experts' assessments.

Fixed-model approaches lack transparency in a way that they impose the model architecture without providing the logic behind it and without describing how the higher-level characteristics are decomposed

to lower-level sub-characteristics and metrics. They also do not provide guidelines on how to use measurement results to evaluate software product quality. Some models seem not to be even consistent as how the characteristics are decomposed. In addition the distinction between particular quality characteristics according to their definitions is not clear. For example, the average developer will not be able to distinguish between characteristics like interoperability [21], adaptability [22], and configurability [24], as they might be regarded as being identical. Define-your-own-model approaches address some of transparency and flexibility weaknesses of fixed-model approaches. They do not impose any prescriptive set of characteristics, so that product-, process- and resource-related characteristics could be combined. Directly-defined models like SQUID [29] provide a description of how to decompose high-level quality characteristics into lower-level sub-characteristics and metrics. However, they say hardly anything about how to compose measurement data and propagate it into quality prediction. The exception is the approach presented in [30] that uses Bayesian Belief Nets (BBNs) to propagate quality assessments from a graph based model.

Indirectly-defined models also cope with combining measures into quality evaluations, however, they face some transparency and flexibility problems. For example, statistical approaches deal with the problem of composing metrics into quality prediction using, for instance, regression equations.

Nevertheless, Ohlson and Alberg [33] claim that character of measurement data only allows ordering modules according to their quality rather than giving objective quality assessments. They also point out that many statistical models that assume normal distribution are applied to model software quality, whereas in many cases such assumptions cannot be made. As far as the reusability of statistical models is concerned, only general conclusions coming from multiple applications of the model are usually reused as guidance within other projects. For instance, Briand and Wuest [32] state that coupling between software modules indicates quality risks, but the same conclusions cannot be made regarding cohesion.

The quantitative character of the input for statistical models limits their application in early stages of the software lifecycle where only few measurement objects are available. Companies that have no measurement programs may experience difficulties with the efficient application of mathematical models in any stage of the software lifecycle.

However, statistical models, unlike directly-defined ones, do cope with redundant and contradicting quality characteristics. The Principal Component Analysis could be employed to identify a non - redundant set of characteristics and metrics. Some more interactive solution could be found in one of the recent approaches to model non –functional requirements: QARCC [20] and NFR -Framework. In those approaches, the user supported by the automated tool identifies overlaps in the graph-based model.

Recent experiments with artificial intelligence approaches have not brought any breakthrough solutions either. Despite the possibility of combining qualitative and quantitative data for more exact early evaluations, their ability to reuse quality experiences across projects is still limited by project similarity. Machine learning models like decision trees or neural networks require a significant amount of training data to achieve satisfactory accuracy of quality estimations. Even then, the result of an evaluation could be of very low accuracy when an evaluated project differs substantially from the past ones. In addition, the structure of a neural network lacks transparency.

An important problem, common to all kinds of quality models, is still the lack of comprehensive guidelines on how to produce a consensus view of quality characteristics and their relationships [41], as well as the inability to reuse quality experiences across different projects and companies to improve the efficiency of quality estimation.

The last issue is tool support. Since quality models should to support software practitioners and minimize quality assurance effort, automated tools are required. Most of the existing quality approaches include dedicated software tools.

In general, there is no consensus yet on how to define and categorize software product quality characteristics. Here we will try to follow as much as possible a standard terminology, in particular the one defined by ISO 9126 [22]. In ISO 9126, a quality *characteristic* is a set of properties of a software product by which its quality can be described and evaluated. A *characteristic* may be refined into multiple levels of *sub-characteristics*.
An *attribute* is a measurable physical or abstract property of an entity. By making a *measurement*, a

measure is assigned to an attribute of an entity, using a metric. A *metric* is the defined measurement method and the measurement scale and the *measure* is the number or category assigned to an attribute.
The Table 1 from [2] compares characteristics of different quality models. The table illustrates the characteristics and their updates during the last 30 years.

Besides the famous ISO 9000, ISO has also release the ISO 9126: Software Product Evaluation: Quality Characteristics and Guidelines for their Use-standard [22] (among other standards). ISO 9126 in the table is based on revision from 1998, which is version between ISO/IEC 9126:1991 and ISO/IEC 9126:2001.
The ISO/IEC 9126:2001 contains 4 parts:
- Part 1: Quality Model
- Part 2: External Metrics
- Part 3: Internal Metrics
- Part 4: Quality in use metrics

Moreover, this document (ISO 9126-1) – Quality Model – contains a two-part quality model for software product quality, that is:
1. Internal and external quality model;
2. Quality in-use model.

The first part of the two-part quality model determines six characteristics in which they are subdivided into twenty-seven sub-characteristics for internal and external quality, as in Figure 1. These sub-characteristics are a result of internal software attributes and are noticeable externally when the software is used as a part of a computer system. The second part of the two-part model indicates four quality in-use characteristics, as in Figure 2.

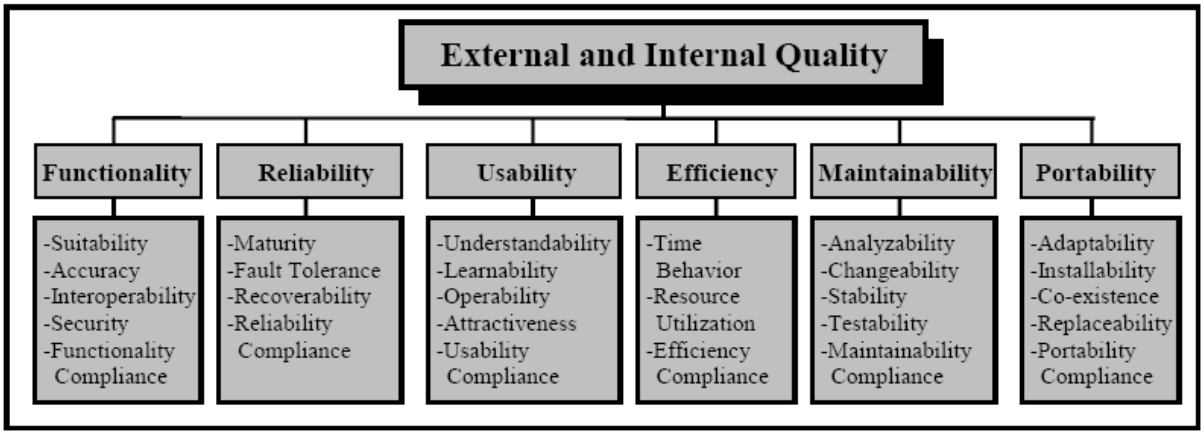

Fig. 1. ISO 9126 quality model for external and internal quality (characteristics and sub-characteristics)

Finally, the fourth document of the ISO 9126 series – quality in-use measures – contains a basic set of measures for each quality in-use characteristic, explanations of how to apply them, and examples of how to use them in the software product lifecycle. The quality in-use measures shown on Fig. 3 are classified by the characteristics defined in ISO 9126-1 and Guidelines for their use [22].

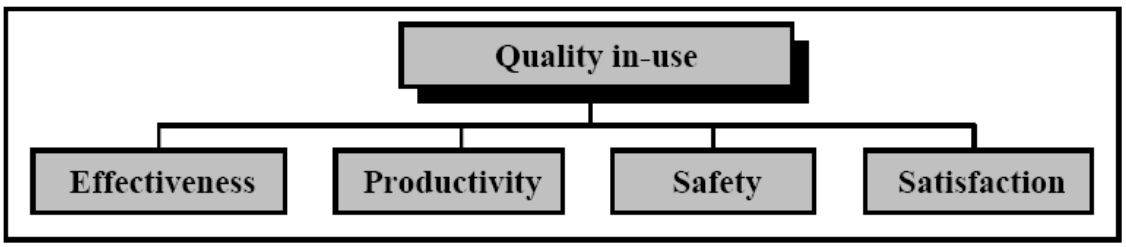

Fig. 2. ISO 9126 quality model for quality in-use (characteristics)

Furthermore, this set of ISO standards could be used by the following intended users during the software development life cycle:
1. Developers.
2. Quality managers.
3. Maintainers.
4. Evaluators.
5. Acquirers.
6. Suppliers.
7.Users

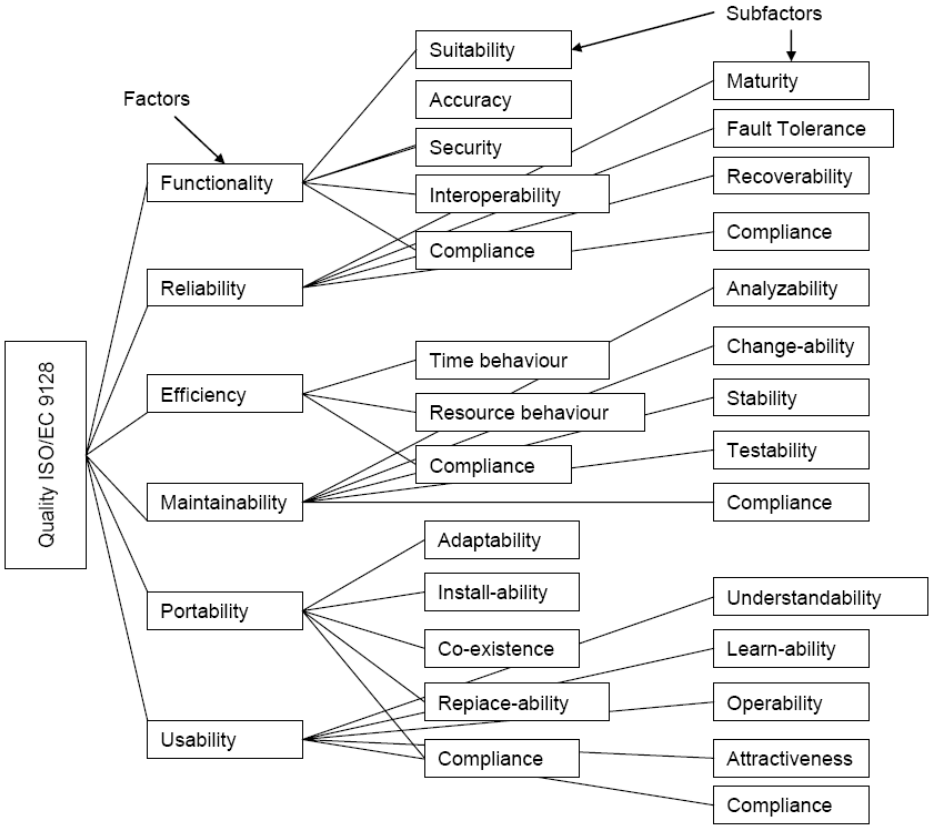

Fig. 3. ISO 9126: Software Product Evaluation: Quality Characteristics and Guidelines for their Use

This paper is going to show that as new domains evolve and are understood there is a need to review our interpretation of quality in those new domains and where appropriate new domain-specific quality factors identified as in a new, fast growing area of Web design. Web site development is maturing from the enthusiastic experimental practice of early years to a more professional discipline, addressing the needs of Web site visitors and owner organizations. Quality is central to this maturing and it is necessary to have a full understanding of the meaning of quality in the context of the ever-changing Web.

In order to understand the quality requirements of a Web site, it is necessary to consider the purpose of Web site software. From a user perspective there is a substantial range of "need-to-include" features, which are appropriate to Web sites.

Web sites need to be easy-to-find, easy-to-download and easy-to-understand. Users need to be confident with the content of the site and with the objectives of the site owner. Web sites need to be interactive and need to incorporate a full range of navigational aids. From an organizational perspective, Web site software is intended to communicate an organizational image and message, to inform visitors to the site, to support access to information and knowledge and to support the sale of products and services through electronic commerce. These objectives for Web site applications are different to those of traditional applications, which generally perform a data processing activity. Consequently, Web sites have different quality considerations.

This paper identifies five new quality factors for the Web (*visibility, credibility, intelligibility, engagibility and differentiation*), together with their characteristics and a checklist of enablers, which can be used by specifiers, designers, developers and evaluators to create quality Web sites. All of these quality factors must be considered as part of a Web site development strategy. However, they were devised prior to the commercialization of the Internet and are more focused towards traditional data processing and information retrieval.

This research shows that they are insufficient for the requirements, opportunities and challenges for both user and organization, which are presented by the changing and active Web. Also, the last decade marked the first real attempt to turn software development into engineering through the concepts of Component-Based Software Development (CBSD) and Commercial Off-The-Shelf (COTS) components. The idea is to create high-quality parts

and join them together to form a functioning system. One of the most critical processes in CBSD is the selection of the COTS components from a repository that meet the user requirements. Current approaches try to propose appropriate quality models for the effective assessment of such components. These proposals define quality characteristics, attributes, and metrics, which are specific to the particular nature of COTS components and CBSD. However, we have found that the information required evaluating those components using those quality models and metrics is not usually available in the existing commercial software repositories. Depending on the projects' similarity level quality model should support the reuse of measurement data as well as quality characteristics and their relationships. On the one hand, reusable quality modeling will reduce the time and cost of quality assurance. On the other hand, it will improve the accuracy and efficiency of quality evaluation, as subsequent experience can contribute to improving existing models. For example, the same model could be reused in every new release of the same software product, and experience from a previous release could be incorporated into an improved model used in the next release. Characteristics of software product lines as well as experience with several existing quality modeling approaches have guided us in defining three main requirements for appropriate quality modeling: *flexibility, reusability, and transparency*. A quality model should be *flexible* because of the context dependency of software quality. There are several quality contexts: company context, project context and process context. On the one hand, *reusable* quality modeling will reduce the time and cost of quality assurance. On the other hand, it will improve the accuracy and efficiency of quality evaluation, as subsequent experience can contribute to improving existing models. For example, the same model could be reused in every new release of the same software product, and experience from a previous release could be incorporated into an improved model used in the next release. A quality model should provide the rationale of how certain characteristics are related to others and how to identify their sub-characteristics. *Transparency* of a quality model also means that the meaning of the characteristics and relationships between them are clearly (unambiguously) defined. People involved in model development and application should understand it in order to gain knowledge from it as well as to identify redundancies or contradictions among quality characteristics. An example of a contradiction could be modularization in object-oriented software. It improves software reliability, but usually at some cost in efficiency. The model should also allow the project stakeholder to directly interfere in the model structure to modify it if needed.

3 Cost of Software Quality (CoSQ)

3.1 Economics of Quality - Literature review

Cost of quality represents any and all costs that organization incurs from having to repeat a process more than once in order to complete the work correctly. Cost of quality (CoQ) is an accounting technique introduced by Juran in 1951 as a means of providing justification to management for investments in process improvements. Cost of software accounting (CoSQ) is useful to enable our understanding of the economic trade-offs involved in delivering good-quality software. Commonly used in manufacturing, its adaptation to software offers the promise of preventing poor quality but, unfortunately, has seen little use to date [3]. Different authors and researcher have used different ways to classify components for quality cost, if we look carefully their understanding about various components are approximately the same.

Pressman [7] has divided the cost of quality into Prevention, appraisal, and failures. As explained by Rex Black [4] that: "Investing in Software Testing, decrease The Cost of Software Quality". He has mentioned costs of quality into two major types: conformance and nonconformance as shown on Fig. 4.

$$CoSQ = Cost_{Conformance} + Cost_{non\text{-}conformance}$$

The definition and categories of quality costs may be given differently by diverse authors. Some use the terms "quality costs", "costs of quality," "economics of quality," "poor quality cost," "price of non-conformance," or "cost of poor quality." The American Society for Quality (ASQ), Quality Cost Committee defined 'quality costs' as a measure of the costs specifically associated with the achievement or non-achievement of product or service quality. The total of the quality costs includes prevention costs of nonconformance to requirements, appraising costs of product or service for conformance to requirements, and failure costs

of products not meeting requirements. As the quality function evolved from inspection (quality control) to more preventive activities (quality assurance), quality cost collection was expanded into prevention, appraisal, and failure costs [(Gryna, 1999; Cartin, 1999 and Campanella, 2003)]. Failure costs are divided into two subcategories: internal and external. Dan Houston [3] has defined Cost of quality in his article "Cost of Software Quality: A Means of Promoting Software Process Improvement" as follows;

$CoSQ = Prevention_{Cost} + Appraisal_{Cost} + Internal\ failure_{Cost} + External\ failure_{Cost}$

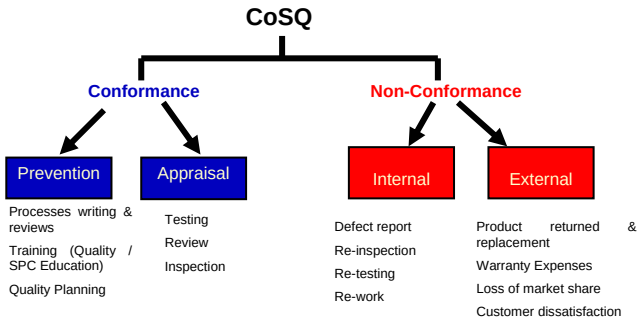

Fig. 4 The Cost of Software Quality

By now we have clear understanding of four components of the Quality cost. With the help of these four components we will discuss the theoretical model suggested by researcher based on the results gathered from the manufacturing industries. Following Fig. 5, is graphical presentation of the CoSQ given by most researchers [2-3], [5-6].

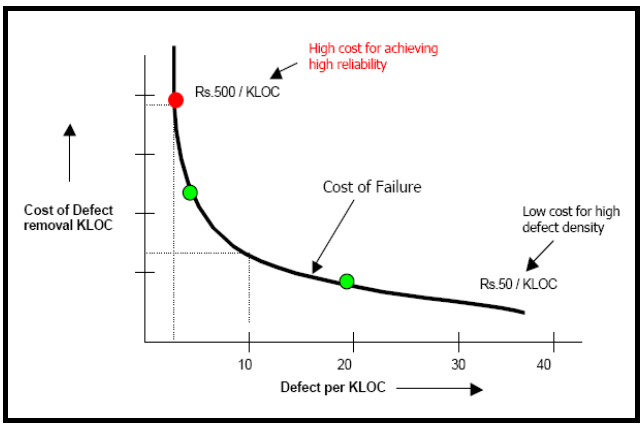

Fig. 5. The cost of high reliability

The above graph is showing that for achieving high reliability, close to red dot (almost zero defect) the cost is very high but achieving a reasonable level (area between two green dots) of quality does not require very high cost. To remove defect after reaching at very low defect density the cost of detection would be very high (Rs.500/KLOC) whereas the defect detection was relatively easy as numbers of defect were high (high defect density) the cost to remove defect is approximately 10 times lesser. Cost mentioned on the graph are imaginary numbers just to give an idea that cost of defect removal at high defect density would be lower and cost at low defect density would be high.

Several studies [42-44] described meanings of these quality cost categories as follows:

• Prevention costs (PC) are those costs associated with quality planning, designing, implementing and managing the quality system, auditing the system, supplier surveys, and process improvements.

• Appraisal costs (AC) are associated with measuring, evaluating, or auditing products, and product materials to ensure conformance with quality standards and performance requirements.

• Failure costs (FC) are those losses associated with the production of a nonconforming product; they can be divided into internal and external.

• Internal failure costs (IFC) are associated with failures and defects of processes, equipment, products, and product materials that fail to meet quality standards or requirements.

• External failure costs (EFC) are generated by defective products, services, and processes during customer use. They include warranties, complaints, replacements or recalls, repairs, poor packaging, handling, and customer returns.

3.2 Statement of Problem

Implementing effective quality cost program has made most companies reduce scraps/rework and costs of poor quality. It also has led to the development of a strategic quality improvement plan consistent with overall organizational goals. Quality cost information is rarely exchanged among businesses. Quality professionals are still trying to determine the main factors and measures aiding in the successful quality cost programs and what problems can be incurred in the quality cost program implementation. The purpose of this study was to identify main factors and measures that aid in the success of the quality costs program and problems

that quality professionals might experience in implementation. The three primary questions are:

Research Question 1: What are the main factors and measures that aid the success of the quality costs program at the surveyed organizations?

Research Question 2: What are problems experienced in implementation of the quality costs program at the surveyed organizations?

Research Question 3: For each of the four categories of the cost of quality (prevention, appraisal, internal failure, and external failure), which category is the highest priority for cost reduction?

In order to answer the research questions, the descriptive data from more than 50 product teams across the industry 630 respondents were summarized and divided into three sections.

3.2.1 Factors and measures

Research question one: What are the main factors and measures that aid the success of the quality costs program at the surveyed organizations?

Many of the teams interviewed in surveyed companies as part of acquisitions. Each of these acquired teams had different level of software engineering maturity and had their own quality processes and measurements. The Standardized Software Quality Assessment model – SSQA [45], as a quality assessment program, is applied in order to find out main factors and measures that aid the success of the quality costs program at the surveyed organizations. The SSQA is a very important component of the company Quality Management System (QMS). QMS is defined in ISO 9001 as "the organizational structure, responsibilities, procedures, processes and resources for implementing quality management necessary to achieve the quality objectives" (see reference #1 for more information on QMS). In every company, quality management has shifted emphasis from merely the reduction of things gone wrong to emphasis on the increase in things done right for the customer. This new emphasis on quality management has fostered an environment of productivity improvement in processes as well as product and service offerings. Equally important, it emphasizes communication, teamwork and employee satisfaction. Feedback from companies' customers was that they appreciate SSQA's program effort and that the adoption of the QMS had improved the communication, the quality and the overall customer satisfaction. The Quality Management System (QMS) that was implemented at surveyed companies is illustrated in Fig. 6. The three key elements of such Quality Management System are:

- Common processes,
- Measures, and
- Continuous improvement

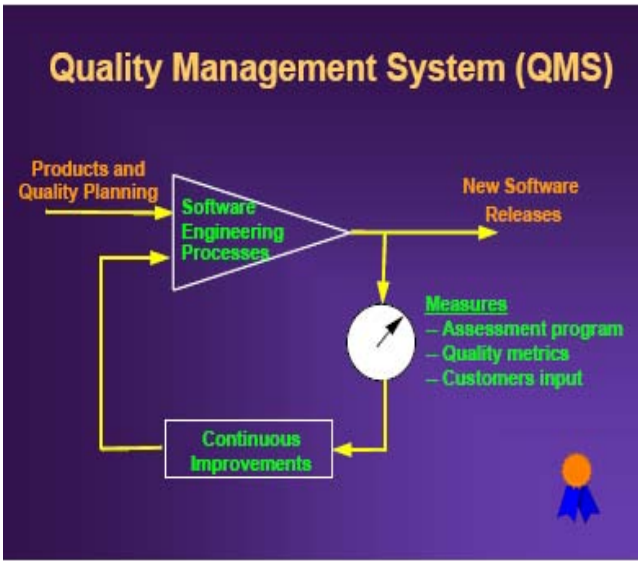

Fig. 6: Quality Management System (QMS)

For every software product introduced, including upgrades of previously released products, all the three elements of the Quality management System are essential for meeting and exceeding customer satisfaction. A key element of the Quality management system is the SSQA Quality Assessment methodology that enables a disciplined quality improvements on a product-by-product basis.

Our quality assessment journey started several years ago, as integral part of the "quality partnerships" that we initiated with our key customers. The first step in deployment of a quality assessment program was to select a suitable quality assessment scheme that addresses the quality and business needs of the company.

We used the following selection criteria:

- Suitable to software industry culture
- Scalability to individual software product teams with varying maturity levels
- Providing a quantitative measure
- Leading to continuous and quantifiable improvements
- Ability to conduct assessments quickly and with minimum effort
- Assessment methodology that support quality partnership with company's customers

We evaluated two general quality assessment schemes - ISO9000 and the Malcolm Baldrige

National Quality Award and three software industry specific assessment schemes – CMM, SPICE/ISO15501, and the semiconductor industry recommended assessment program – SSQA (Standardized Software Quality Assessment). We reviewed our selection criteria with companies' customers and the final recommendations were to select the SSQA as the most suitable assessment vehicle in software industry by which different product teams can evaluate and improve the maturity and effectiveness of their software development practices. In addition, we extended the standard SSQA methodology to include our own methodology for periodic mini-assessments that are very important in driving continuous quality improvements by the different product teams.

The two parts of the company assessment program are:

• *Full Assessments:* Base-line quality assessments of each product team (interview 10-15 engineers from R&D, QA and Management). For the base-line assessments we use the standard SSQA methodology

• *Mini-Assessments:* Periodic quality reviews with each product team - Driving continuous quality improvements. For the mini-assessments we use our own methodology which is an extension to the standard SSQA methodology.

We use software quality assessments at the product level to determine the current maturity level of the product software development engineering practices; to foster quality improvement; to share "best practices;" and to ensure that the product software development and support processes are effective in achieving customer satisfaction.

Base-line SSQA Assessments: Product base-line SSQA assessments are conducted once per product. The intent of base-line assessments is to determine the base line status and scoring of the software development processes, to point out strengths, and to identify opportunities for improvement. Using the SSQA methodology, the Assessment Review Team interviews across section of the product team engineers, and collects evidence on the processes that have been deployed in the management, development, rollout, and support of the software product. During the review process the assessment team compares the current quality system to a "perfect" or ideal quality system, as described in the SSQA 12-element guidelines. These 12 elements are listed below. The assessment team scores the current software maturity level each of the 12 quality elements, evaluating each on four categories: management commitment, approach, deployment and results. Taken together, the 12 assessment elements provide a comprehensive evaluation of the product team commitment to software quality and customer satisfaction.

The 12 SSQA assessment elements

1. Planning Process
2. Specifications and Reviews
3. Coding Practices
4. R&D Testing
5. Regression Suites
6. Alpha Testing
7. Beta Testing
8. Entry/Exit Criteria
9. User and Training Documents
10. Bug Management
11. Support Services
12. Customer Feedback

Base-line quality assessments are conducted to review the software engineering maturity levels of different product teams. Typically, the assessment review team interview 10-15 members of each product team, representing Management, R&D, Marketing, Application Engineers, Tech Pub and Operations.

The "checklist" of subjects that we used in the interviews is shown below:

Checklist used in interviews:

Product Life-Cycle

•Establish basis - product overview, goals, team,...
•Requirements - defined , reviewed, changed ….
•Plans & progress monitoring
•Functional & design specs, coding, unit tests
•Test process, Alpha, Beta
•Rollout planning
•Phase hand-off criteria, release criteria ….
•Release coordination ….

Customer support

•Defect management process
•Communicating with customers
•Metrics, response time, backlog ….

Support Systems

•Management support
•Staffing, skill, and training
•Quality goals
•Code Reviews
•Customer interactions / feedback
•Release Management
•QA / customer advocate

•Configuration Management
•Tech publications
•Computing Resources & Backup

During the interviews, the assessment team verify deployment against the company software development life cycle process. Following each interview, the assessment review team provide detailed report to each interviewee on his/her interview finding – including strengths and suggested areas for improvements.

The SSQA methodology also includes a detailed guide to be used by the Assessment Review Team to determine objectively the scoring levels for the 12 SSQA elements. A score ranging from 0 (poor) to 10 (outstanding) identifies the maturity level of each quality element, including process definition, process deployments, results obtained, priorities, and impact on the customers.

• Level 0 - No systematic approach apparent

• Level 1-2 - Beginning of a process in place; although decentralized and fragmented

• Level 3-4 - Process direction is being defined; more centralized and less fragmented

• Level 5-6 - Significant effort underway. Deployment in major areas. Some results being realized.

• Level 7-8 - Effective quality system fully in place. Significant, positive results. All areas involved.

• Level 9- 10 - Setting the standard for achieving Total Customer Satisfaction.

Periodic Mini-Assessments:

In every company, one of the main goals of the quality assessment program is to drive continuous quality improvements by the different product teams. For this reason, we extended the standard SSQA methodology for full-assessments of the base-line quality maturity levels of different product teams, to include quick methodology for periodic mini-assessments that are very important in increasing quality awareness and in driving continuous quality improvements by the different product teams.

Typically, every 6-12 months, we conduct for each product a 2-hours mini-assessment review with the product "core team" – 6 to12 key people of the product team that represent the R&D, the Marketing and the Application Engineering functions of the product team. The "template" questions that are covered in the mini-assessments reviews are shown below. Also, for the scoring of the mini-assessments we use our company "standard" scoring, which is a scoring level between –2 to +2 (+2 = very satisfied, +1 = satisfied, 0 = neutral, -1 = weak, -2 = very weak) that is used to measure the process maturity, results obtained, and the impact on customers.

Sample Results

Base-line quality assessments are conducted to determine the base line software engineering maturity level for each product. The assessment includes a review of the status of the software development processes, looking at strengths, and identifying opportunities for improvement.

The assessment team scores the software maturity level each of the 12 quality elements, evaluating each on four categories: management commitment, approach, deployment and results. Taken together, the 12 assessment elements provide a comprehensive evaluation of the product team commitment to software quality and customer satisfaction. A numerical score ranging from 0 (poor) to 10 (outstanding) is given to each of the 12 quality elements. The total scoring for the product is an average of the scorings for the 12 elements.

At the end of the base-line assessment, an assessment report is presented to the product team and their management. The report provides a macro view of the state of the quality system, recognize achievements, point out shortcomings and opportunities, and offers recommendations. Periodic Mini-Assessments, are conducted, typically, every 6-12 months, we conduct for each product a 2-hours mini-assessment review with the product "core team" – 6 to12 key people of the product team that represent the R&D, the Marketing and the Application Engineering functions of the product team. The main goal is to drive continuous quality improvements by the different product teams. For each of the 12 elements, we summarize the status, strengths and improvement opportunities that were recommended by the product core team. In addition, the core team recommends the scoring.

The mini-assessment report also includes a list of the key improvement activities that the core team agreed to carry out. An example of such a list is shown below: Continuous improvement plan for next 6 months: (1) # of Bugs: Reduce WDC of backlog by 25% (2) Vigilance on coding errors: # of Lint errors - target: 0 fatal warnings at code freeze. # of Purify errors - target: 0 at code freeze (3) Regression Suite: Increase line coverage (PurCov) by 15% (4) # of undocumented error messages: Reduce # of undocumented messages from 30% to 20%, etc.

3.2.2 Problems in quality costs implementation

Research question two: What are problems that you experienced in implementation of the quality costs program at your organization?

The answers from participants were varied and can be summarized into four groups: measurements, people, process, and information. Each group has components that caused unsuccessful quality cost programs in the manufacturing environment.

1) Measurements include lack of an appropriate system and incorrect methods of collecting quality cost categories.

2) People issues were lack of support from the senior leadership team, and lack of cooperation from the accounting and finance departments, and managers and employees who are deficient in knowledge of 'Cost of Quality' and training.

3) Process contains inconsistency from plant to plant and ineffective process standards.

4) Information indicates lack of clear instruction and inadequate information to properly design and process.

Furthermore, several respondents indicated that a culture that favors correction over prevention also led to an unproductive quality cost program.

3.2.3 Quality cost reduction

Research Question three: For each of the four categories of the cost of quality (PC, AC, IFC, and EFC), which category do you think is the highest priority for cost reduction?

A number of respondents expressed that PC and EFC were the highest priority for cost reduction. They agreed that prevention costs provided tools and training for reducing wastes in the process. Among forty respondents who answered this question, fifteen voted for EFC, fourteen for PC, nine for IFC, and two for AC.

These factors, measures and problems identified from the survey results were then used in developing an empirical model of quality costs to assess the quality management systems in the manufacturing environment.

The procedure for using the economics model to compare defect-detection techniques starts with the compilation of a list of all the faults that were found by the defect detection techniques (DDT) [12]. They are simply numbered and enriched with additional information. This information should at least be estimations of the MTTF, the severity, and the change effort. The latter is divided into the specific values for each period. Having this list, the faults can be assigned to the set of internal and external failures for each technique.

This can be used to calculate the costs regarding internal and external failures for each technique in order to optimize software quality cost.

The appraisal costs are also needed to get the complete cost calculation. We therefore calculate the appraisal costs by adding at least tool and personnel costs. Some of this information is available from defect databases and accounting. The remaining values have to be estimated by experts. Especially for the revenues estimating is the only possibility.

We do not have hard data about the cost savings because the failures do not occur. However, they cannot be ignored because they are the main benefits of using defect-detection techniques. Therefore estimates are important and necessary for the quality economics of DDTs [12]. The comparison can finally be based on the costs only or on further metrics such as in the IOSTP [8,9]. Test activities across the project include *Sequential usage of* defect detection techniques.

If defect-detection techniques are analyzed that are used one after the other, the procedure has to be slightly changed. The defects that were revealed and removed before a technique is used cannot be counted as external failures for that technique. Only faults found after the technique under investigation are external failures.

As mentioned above this blurs the data for all techniques because if an extremely effective technique was used that found a lot of defects that were removed before the next technique is applied, the next one cannot be as effective as it normally would be with all faults still in the software. However, this is an inherent problem. What we can do is to experiment with different orders of the techniques and with different amounts of effort spent for each technique. This way we can find the optimal combination of techniques [10]. An example can be found in the following sections.

The costs of achieving quality and the costs due to lack of quality have an inverse relationship to one another: as the investment in achieving quality increases, the costs due to lack of quality decrease. This theoretical model is shown below in Fig. 7. This shows that as appraisal and prevention cost increases, the failure cost will decrease until an optimum point is reached. After this optimum point, the increase in appraisal will not be offset by the decreased in failure cost. Researcher have noticed

that in the initial phase appraisal measures cause internal failure to increase as these measures detect more errors at early stages, but error removal at early stage is much cheaper compare to error removal at later stage. But overall appraisal activities decrease external failure as a result total failure decreases. A small increase in prevention measures will normally create a major decrease in total quality cost.

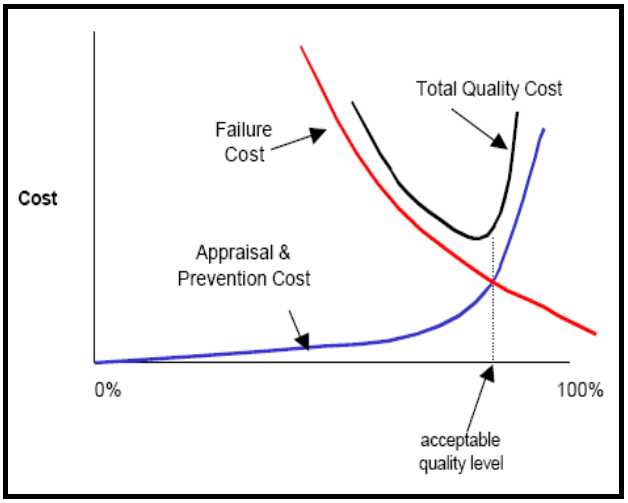

Fig. 7. Model of software quality

3.3 Quality Cost Analysis

The objective of the quality cost analysis is not to reduce the cost, but to make sure that the cost spent are the right kind of cost and that maximize benefit derived form that investment. Traditional view of the cost of quality revolved around failure related activities. Due to quality cost analysis the major emphasis has been shifted to prevention and appraisal. As we all know that corporate understand the language of money, quality cost analysis emerged the concept of studying quality related cost as means of communication between the quality staff department and company managers. Challenge is how do you go about taking economic considerations into account when designing or modifying a system?

- How do you account for the costs involved?
- How can costs and benefits be "traded-off" against quality attributes or functionality?

A cost benefit analysis is done to determine how well, or how poorly, a planned action will turn out. Although a cost benefit analysis can be used for almost anything, it is most commonly done on financial questions. Since the cost benefit analysis relies on the addition of positive factors and the subtraction of negative ones to determine a net result, it is also known as running the numbers.

A cost benefit analysis finds, quantifies, and adds all the positive factors. These are the benefits, and then it identifies, quantifies, and subtracts all the negatives, the costs. The difference between the two indicates whether the planned action is advisable. The real trick to do a cost benefit analysis well is making sure you include all the costs and all the benefits and properly quantify them.

The key consideration in any analysis of the cost of quality is visibility. It is the visibility gained form cost of quality analysis that enable the QA people involved to focus their attention on those activities which discover, and correct the root cause of the software defects. This root cause analysis allows the QA people to determine how the development process can be improved to prevent further defects. Following Fig. 8, is the graph, that is showing the theoretical model of CoSQ, adopted form Knox [3]. CoQ is a proven technique in manufacturing industries both for communicating the value of quality initiatives and for indicating quality initiative candidates. CoSQ offers the same promise for the software industry, but has seen little use to date. Initial uses of CoSQ show that it can be a very large percentage of development costs, 60 percent or higher for organizations which are unaware of improvement opportunities. CoSQ has demonstrated its value in measuring the ROI of a software improvement program across the software industry.

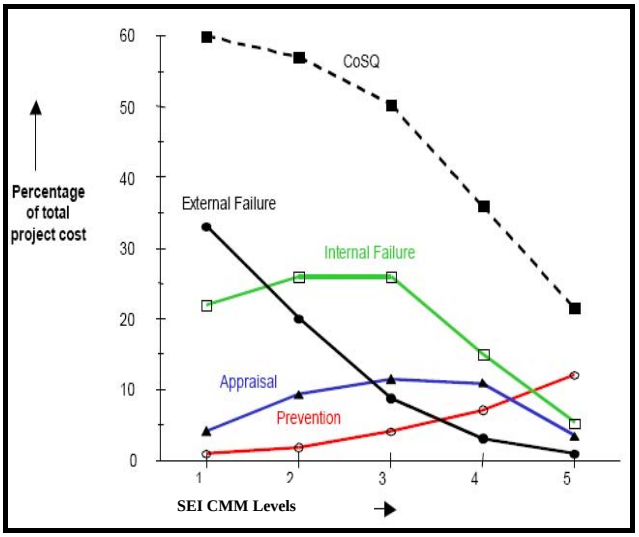

Fig. 8. Knox's Theoretical CoSQ Model for CMM Levels Model [3]

Starting with a the total CoSQ (TCoSQ) at 60% of development costs (based on two industry figures)

for CMM level 1 organizations, Knox used manufacturing experience to hypothesize that CMM level 5 organizations can cut this CoSQ by about 67%. He then rationalized the four component costs at each CMM level. His model suggests that for level 3 organizations, CoSQ is about half of development costs.

3.4 Defect Potentials and Defect Removal Efficiency Analysis

There are two very important measurements of software quality that are critical to the industry:

1. Defect potentials
2. Defect removal efficiency

All software managers and quality assurance personnel should be familiar with these measurements because they have the largest impact on software quality, cost, and schedule of any known measures.

The phrase *defect potentials* refers to the probable numbers of defects that will be found during the development of software applications. As of 2008, the approximate averages in the United States for defects in five categories, measured in terms of defects per function point and rounded slightly so that the cumulative results are an integer value for consistency with other publications by the Capers Jones [11], follow.

Note that defect potentials should be measured with function points and not with lines of code. This is because most of the serious defects are not found in the code itself, but rather in requirements and design. Table 2 shows the averages for defect potentials in the U.S. circa 2008.

Requirements defects	1.00
Design defects	1.25
Coding defects	1.75
Documentation defects	0.60
Bad fixes	0.40
Total	**5.00**

Table 2 Averages for Defect Potential [11]

The measured range of defect potentials is from just below two defects per function point to about 10 defects per function point. Defect potentials correlate with application size. As application sizes increase, defect potentials also rise. A useful approximation of the relationship between defect potentials and defect size is a simple rule of thumb: application function points raised to the 1.25 power will yield the approximate defect potential for software applications. Actually, this rule applies primarily to applications developed by organizations at Capability Maturity Model®(CMM®) Level 1. For the higher CMM levels, lower powers would occur. Reference [11] shows additional factors that affect the rule of thumb.

The phrase *defect removal efficiency* refers to the percentage of the defect potentials that will be removed before the software application is delivered to its users or customers. As of 2007, the average for defect removal efficiency in the U.S. was about 85 percent. If the average defect potential is five bugs – or defects – per function point and removal efficiency is 85 percent, then the total number of delivered defects will be about 0.75 per function point. However, some forms of defects are harder to find and remove than others. For example, requirements defects and bad fixes are much more difficult to find and eliminate than coding defects. At a more granular level, the defect removal efficiency against each of the five defect categories is approximate in Table 3.

Defect Origin	Defect Potential	Removal Efficiency	Defects Remaining
Requirements defects	1.00	77%	0.23
Design defects	1.25	85%	0.19
Coding defects	1.75	95%	0.09
Documentation defects	0.60	80%	0.12
Bad fixes	0.40	70%	0.12
Total	**5.00**	**85%**	**0.75**

Table 3 Defect Removal Efficiency [11]

Note that the defects discussed in this section include all severity levels, ranging from severity 1: *show stoppers*, down to severity 4. Obviously, it is important to measure defect severity levels as well as recording numbers of defects.

4 Software Testing Economics - Hypothetical Case Study

According to the National Institute of Standards and Technology (NIST), eighty percent of the software development costs of a typical project are spent on identifying and fixing defects. With today's executive mandates for speed and agility, such expense to repair is not only unnecessary, it borders on corporate irresponsibility and demonstrates a lack of ability to align IT processes with overall business goals. Software Quality Optimization™ (SQO™) is a forward-thinking approach to software quality that integrates people, processes and technologies toward one specific goal: it ensures that software deployment is synchronized with business goals to

achieve competitive advantage. SQO is a continuous, iterative process throughout the application lifecycle resulting in zero-defect software that delivers value from the moment it goes live.

In this paper, we will:

• Explore the true costs of software defects and their impact on application performance

• Challenge the traditional philosophy that "testing equals quality," and demonstrate how quality processes implemented throughout the application lifecycle can result in measurable performance improvements

• Share seven best practices for optimized application quality and identify the four steps to implement optimized software quality processes

• Provide a definition for a quality optimization platform that drives quality efficiencies across the enterprise, and show you how investing in such a platform can help your organization minimize the costs associated with application development and realize the full potential from your application investments.

4.1 The Real Cost of Software Defects

It is obvious that the longer a defective application evolves the more costly it is to repair. But how much more? The answer might surprise you. According to the CTO of one software development organization, a bug that costs $1 to fix on the programmer's desktop costs $100 to fix once it is incorporated into a complete program, and many thousands of dollars if it is identified after the software has been deployed in the field [46], as described on Fig. 9. Barry Boehm, one of the industry's leading experts on software quality, has published several studies [47,48] over nearly three decades that demonstrate how the cost for removing a software defect grows exponentially for each downstream phase of the development lifecycle in which it remains undiscovered. Since the original study, Boehm's results have been confirmed in a number of subsequent studies [7,11,49]. Further, another major research project conducted recently by the United States Department of Commerce, National Institute of Standards and Technology showed that in a typical software development project, fully 80% of software development dollars are spent correcting software defects. The same NIST study also estimated that software defects cost the U.S. economy, alone, $60 billion per year [49]. Many organizations view the software development lifecycle as a linear process with discrete functions: design, develop, test and deploy. In reality, the software development lifecycle is a cyclical function with interdependent phases. Quality assurance has a role in every phase of that lifecycle, from requirements review and test planning, to code development and functional testing, to performance testing and on into production.

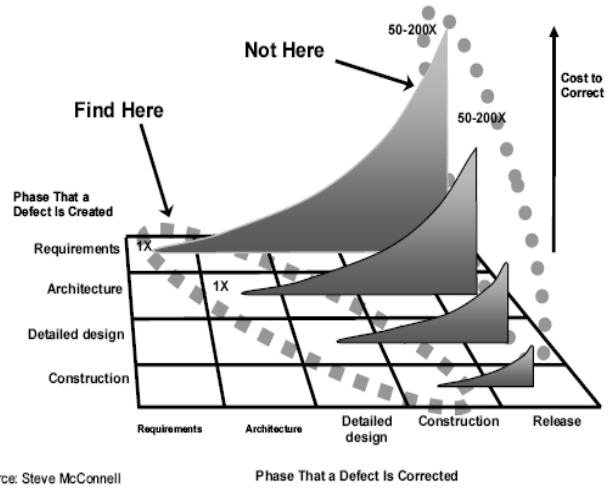

Fig. 9 Engineering Rules for Cost Of Defect Removal [46]

It was unanimously agreed that quality and quality assurance is more than strictly testing at the end of the development process. Starting quality initiatives early and paying attention to quality throughout the development, deployment and production effort is key in order to achieve a baseline goal of zero-defect software.

Testing Equals Quality?

The days when a quick testing phase was squeezed between end of development and deployment date – if it was done at all – are long gone. So is the idea that quality can be "tested into" software. Quality assurance as an overarching concept embracing all aspects of quality management is now a firmly entrenched part of software development practices by best-of-breed companies. The quality concept goes significantly beyond testing. Software Quality Optimization is a holistic approach to everything an organization does throughout the software application lifecycle. Quality frameworks such as the Capability Maturity Model® Integration

(CMM®/CMMI®) and the ISO® 9001 Quality Management System embody total quality concepts by suggesting processes that define and document every aspect of software development for every phase of the life cycle. Depending on the size of the organization and the industry segment, these quality frameworks may be a perfect fit or may need to be tailored to suit the organization. Regardless of the chosen approach, quality should be the mantra for any organization that values its customers and is accountable to the business needs of the enterprise. "Test first" concepts are the cornerstone of the newly emerging Agile development paradigm [5] but there is more to holistic quality management than just testing. Quality principles such as peer reviews and design for testability can be applied throughout the development phases from requirements specifications to design and code. Along the way, increasingly powerful quality automation solutions can be deployed to support the quality initiative. Testing early may seem to be an oxymoron to some people. After all, you need code before you can test. Or do you? The Agile development community has turned traditional beliefs about testing upside down with their slogan "test first". Of course you cannot test software in the conventional sense when no code is written. But other things can be tested. For example, requirements specifications can be "tested" for completeness before any development begins. Close collaboration between testers, programmers, and requirements analysts in structured reviews of the specifications provides a mechanism for detecting and correcting defects in these upstream work products. This is most effective when the users are involved directly with development and quality personnel. Early collaboration allows test plans and test cases to be developed in parallel during the specification phases. Apart from the obvious benefit of having these materials ready before the first testable code emerges and thereby accelerating the testing phase, there are other, more subtle benefits. In particular, developing test plans early on can show logic flaws in the design before coding starts, which avoids unnecessary rework later on. It can also show that certain parts of the software are difficult if not impossible to test exhaustively. Design for testability is therefore encouraged and makes later testing much easier.

An environment that is optimized for software quality helps you to [12]:

Develop more efficiently and produce a higher quality product because you

• Eliminate defects at the time they are introduced into the engineering process

• Improve staff utilization by freeing them up to focus on new functionality, rather than dealing with unplanned rework or fixing defects

• Reuse test assets and eliminate duplicated efforts

• Significantly reduce the risk of project failure

Deploy faster and significantly reduce ongoing maintenance costs because you

• Integrate quality throughout the lifecycle, and therefore eliminate late project surprises that impact software release schedules

• Enable collaboration between personnel responsible for various aspects of quality, leveraging quality assets from earlier quality phases to downstream phases

• Reduce error turnaround time by finding errors sooner

• Are able to release a zero-defect application and therefore benefit from the tremendous savings associated with not having to repair at the back-end

Improve customer satisfaction and build competitive advantage because you

• Become a more nimble organization and are able to more quickly respond to changing customer needs and the underlying business processes that support them

• Accelerate go-live timelines and improve your organization's time-to-market value

Optimizing software quality is driven by two important items: selecting the most appropriate best practices to help an organization achieve its specific business goals and utilizing a platform capable of supporting those best practices. Quality practices can include peer reviews, test planning and test case development. Strong, overall test management and metrics can be used to monitor application quality and the effectiveness of testing activities. Test automation, test management and application performance management software solutions can significantly support an organization's quality optimization goals by introducing significant labor efficiencies through test asset reuse and repeatability of test results, as well as providing visibility into quality metrics and enabling the cross-departmental collaboration necessary between all personnel responsible for the various aspects of quality. Fortunately, there is a substantial body of knowledge describing the most highly valued best

practices for software quality assurance in pre-production and production environments. The most direct approach for optimizing quality practices is to leverage this body of knowledge, selecting those practices that most directly support quality optimization goals.

Gone are the days when quality could be viewed as an "after thought", something to be fit in at the end of a development project - if there was any time left. Now, software quality optimization is viewed as strategically important by forward thinking companies. These organizations realize that an emphasis on quality does have a significant return on investment (ROI) attached to it, resulting from reduced cycle times, quicker time to market, efficient use of quality resources and lower ongoing cost of maintenance. Success depends on the integration of early, continuous and collaborative quality practices throughout the application lifecycle. Additionally, and not insignificantly, brand image and customer satisfaction are enhanced with deployment of high quality applications.

Seven of the most effective quality best practices are provided below.

1. Incorporate peer reviews into your software development lifecycle. Industry experts have demonstrated that peer reviews can remove an impressive 60-90% of software defects in the lifecycle phase in which they were introduced [5,7,11,12]. This statistic, alone, makes peer reviews the number one software quality practice to adopt. Through peer reviews, software specifications and source code are reviewed by various members of the software development and quality assurance team to identify any defects before the data can be incorporated into any executable code.

2. Incorporate proven development methodologies that embrace quality activities early in the development process [7,12]. There are many development methodologies, such as Agile, that emphasize testing as an on-going, integral part of the development process. As mentioned previously, numerous studies have proven that defects found early in the software application lifecycle are far less expensive to fix than those found at later stages. Though an older methodology, the V-model is another example of a tried and true development approach that infuses testing into the development process. The V-model for software quality is a straightforward and simple model that shows test activities beginning in parallel with the corresponding development activities. Models such as Agile and V-model dictate that organizations begin planning testing activities and developing preliminary test cases in parallel with the corresponding development phases. In the early requirements and design phases, tests can't be executed against software yet, but conceptual (implementation-independent) test cases can still be developed based on the requirements/designs and used to find errors, ambiguities and omissions in development specifications.

3. Formalize test planning [5,7,12]. Writing formal test plans and test case documentation provides the basis for the most effectively focused and repeatable testing activities. Without these plans and test cases, testing becomes a haphazard, ad hoc activity in which defects - when discovered - may not be reproduced. Worse, their correction usually cannot be verified. Good test plans reflect a testing strategy that ensures efforts are focused on the highest business priority and highest risk areas first, while providing for adequate coverage of the application requirements.

4. Build reusability into regression testing [5]. Regression testing is the practice of testing the code of an application to make sure changes to the application have not broken any existing functionality or negatively impacted performance. Regression test suites are most commonly built up over time from test cases created for previous product releases. Regression test frameworks should be developed with an eye on ease-of-maintenance and reuse. Reusability in the regression testing area is a significant driver of ROI obtained through test automation.

5. Make the investment in test automation software [5,7,11,12]. Taking advantage of test automation solutions reduces time-consuming, effort-intensive manual testing, especially where sets of tests need to be run repetitively many times over or require a lot of resources. The former is especially the case with regression testing, and in the late stages of a development project when the QA and development teams are trying to stabilize builds in preparation for product release. In addition to labor savings, deployment risk can be further reduced by having a documented, repeatable process that can verify test results.

6. Test for quality attributes [5,7,9-12]. Quality attributes is the generic term for a range of software requirements that describe the overall quality behavior of the system, rather than the features and functions it implements. The most common quality attributes include accuracy, performance, stability,

availability, scalability and usability. Testing for the most important quality attributes avoids the nasty surprises that too frequently occur when a functionally correct application is deployed into customer environments, but then fails to perform as expected.

7. Take control of test and application performance management [5,7,9-12]. The effective management of testing activities and the monitoring of key quality metrics can greatly aid the delivery of applications that satisfy their quality objectives. Test management as a best practice ensures provision of enough test coverage of all functional areas of the application based on their relative priorities. Monitoring defect metrics can provide enlightening information about the quality status of the application under development. A concentration of defects in particular code modules can highlight poor architecture/design, while monitoring defect resolution trends and defect densities can provide valuable criteria for determining when an application is ready for release.

Organizations with defect potentials higher than seven per function point coupled with defect removal efficiency levels of 75 percent or less can be viewed as exhibiting professional malpractice. In other words, their defect prevention and defect removal methods are below acceptable levels for professional software organizations. Most forms of testing average only about 30 to 35 percent in defect removal efficiency levels and seldom top 50 percent. Formal design and code inspections, on the other hand, often top 85 percent in defect removal efficiency and average about 65 percent.

As can be seen from the short discussions here, measuring defect potentials and defect removal efficiency provide the most effective known ways of evaluating various aspects of software quality control. In general, improving software quality requires two important kinds of process improvement: 1) defect prevention and 2) defect removal.

The phrase *defect prevention* refers to technologies and methodologies that can lower defect potentials or reduce the numbers of bugs that must be eliminated. Examples of defect prevention methods include joint application design, structured design, and also participation in formal inspections.

The phrase *defect removal* refers to methods that can either raise the efficiency levels of specific forms of testing or raise the overall cumulative removal efficiency by adding additional kinds of review or test activity. Of course, both approaches are possible at the same time.

In order to achieve a cumulative defect removal efficiency of 95 percent, it is necessary to use the sequence of optimum combination of software defect detection techniques (DDT) choices for every software development phase that maximize all over Defect Detection Effectiveness [12].

There are large ranges in terms of both defect potentials and defect removal efficiency levels. The *best in class* organizations have defect potentials that are below 2.50 defects per function point coupled with defect removal efficiencies that top 95 percent across the board. Defect removal efficiency levels peak at about 99.5 percent. In examining data from about 13,000 software projects over a period of 40 years, only two projects had zero defect reports in the first year after release. This is not to say that achieving a defect removal efficiency level of 100 percent is impossible, but it is certainly very rare.

From an economic standpoint, combining formal inspections and formal testing will be cheaper than testing by itself. Inspections and testing in concert will also yield shorter development schedules than testing alone. This is because when testing starts after inspections, almost 85 percent of the defects will already be gone. Therefore, testing schedules will be shortened by more than 45 percent.

Measuring the numbers of defects found during reviews, inspections, and testing is also straightforward. To complete the calculations for defect removal efficiency, customer-reported defect reports submitted during a fixed time period are compared against the internal defects found by the development team. The normal time period for calculating defect removal efficiency is 90 days after release.

As an example, if the development and testing teams found 900 defects before release, and customers reported 100 defects in the first three months of usage, it is apparent that the defect removal efficiency would be 90 percent.

Unfortunately, although measurements of defect potentials and defect removal efficiency levels should be carried out by 100 percent of software organizations, the frequency of these measurements circa 2008 is only about five percent of U.S. companies. In fact, more than half of U.S. companies do not have any useful quality metrics at all. More than 80 percent of U.S. companies, including the great majority of commercial software vendors, have only marginal quality control and are much lower than the optimal 95 percent defect removal efficiency level. This fact is one of the reasons why so many software projects fail completely or experience massive cost and schedule

overruns. Usually failing projects seem to be ahead of schedule until testing starts, at which point huge volumes of unanticipated defects stop progress almost completely.

As it happens, projects that average about 95 percent in cumulative defect removal efficiency tend to be optimal in several respects. They have the shortest development schedules, the lowest development costs, the highest levels of customer satisfaction, and the highest levels of team morale. This is why measures of defect potentials and defect removal efficiency levels are important to the industry as a whole; these measures have the greatest impact on software performance of any known metrics.

Additionally, as an organization progresses from the U.S. average of 85 percent in defect removal efficiency up to 95 percent, the saved money and shortened development schedules result because most schedule delays and cost overruns are due to excessive defect volumes during testing. However, to climb above 95 percent defect removal efficiency up to 99 percent does require additional costs. It will be necessary to perform 100 percent inspections of every deliverable, and testing will require about 20 percent more test cases than normal [11]. Industry data about defect potentials and for Defect Removal Efficiency depends on SEI CMM level as shown on Fig. 10 [11].

4.2 Techniques to analyze return on the testing investment (ROI)

4.2.1 Financial ROI

From a developer's perspective, there are two types of benefits that can accrue from the implementation of good software quality practices and tools: money and time. A financial ROI looks at cost savings and the schedule ROI looks at schedule savings.

Direct financial ROI is expressed in terms of effort since this is the largest cost on a software project. There are a number of different models that can be used to evaluate financial ROI for software quality.

The first is the most common ROI model. We will show that this model is not appropriate because it does not accurately account for the benefits of investments in software projects. This does not mean that that model is not useful (for instance, accountants that we speak with do prefer the traditional model of ROI), only that we will not emphasize it in our calculations.

We subsequently present the second model which we argue is much more appropriate. The models here are presented at a rather conceptual level. We also look at ROI at the project level rather than at the enterprise level. ROI at the enterprise level (or across multiple projects) requires a slightly different approach which we will not address directly here.

Industry data – SEI Levels

CMM Approach		
Measure	Average defects/ function points	DRE [%]
Typical defect potential and delivered defects for SEI CMM Level 1	5.0 Injected .75 delivered	85%
Typical defect potential and delivered defects for SEI CMM Level 2	4.0 Injected .44 delivered	89%
Typical defect potential and delivered defects for SEI CMM Level 3	3.0 Injected .27 delivered	91%
Typical defect potential and delivered defects for SEI CMM Level 4	2.0 Injected .14 delivered	93%
Typical defect potential and delivered defects for SEI CMM Level 5	1.0 Injected .05 delivered	95%

Fig 10 Industry data - Engineering Rules for Defect Removal Efficiency [11]

The most common ROI model, and that has been used more often than not in software engineering, is shown below:

$$ROI_1 = \frac{Total \cdot CoQ \cdot Saved - Test \cdot Investment}{Test \cdot Investment} \quad (1)$$

This ROI model gives how much the Total Cost of Quality (CoQ) savings gained from the project were compared to the initial investment. Let us look at a couple of examples to show how this model works.

Let's use a hypothetical case study to illustrate the use of this cost of quality technique to analyze return on the testing investment. Suppose we have a software product in the field, with one new release every quarter. On average, each release contains 1,000 "must-fix" bugs—unacceptable defects—which we identify and repair over the life of the release. Currently, developers find and fix 250 of those bugs during development, while the customers

find the rest.

Suppose that you have analyzed the costs of internal and external failure. Bugs found by programmers costs $10 to fix. Bugs found by customers cost $1,000 to fix. We analize three cases of software development and testing process which provide *Low Quality, Good Quality and High Quality Results.*

Case 1: Low Quality Results

Case 1 is assumed to be a fairly small systems software project of 251 function points in size. Defect potentials are derived by raising the function point total of the application to the 1.25 power, which results in a total of 1,000 defects or 4 defects per function point [11]. Defect removal efficiency is assumed to be 75% overall. The development team is assumed to be below level 1 on the CMM scale in Software Development Process (SDP) which is unpredictable and poorly controlled ie. Ad hoc level.

As shown in the "**Case 1 Testing**" column in Fig. 5, our cost of quality is three-quarters of a million dollars. It's not like this $750,000 expenditure is buying us anything, either. Given that 750 bugs escape to the field, it's a safe bet that customers are mad!

Case 2: Good Quality Results

Case 2 is exactly the same size and the same class of software as Case 1. The project management desided to improve software testing process (STP) and invested in testing staff $60,000 and test infrastructure $10,000 as shown in the "**Case 2 Testing**" column in Fig. 11.

The development team is assumed to be level 1 on the CMM scale. Defect removal efficiency is assumed to be 85% overall. Defect removal operations consist of six test stages: 1) unit test, 2) new function test, 3) regression test, 4) integration test, 5) system test, and 6) external Beta test.

Case 3: High Quality Results

Case 3 is exactly the same size and the same class of software as Case 1. The development team is assumed to be higher than level 3 on the CMM scale. By means of more effective defect prevention such as Quality Function Deployment (QFD) and Six-Sigma the defect potentials are lower. Defect removal efficiency is assumed to be 95%. Defect removal operations consist of nine stages: 1) design inspections; 2) code inspections; 3) unit test, 4) new function test, 5) regression test, 6) integration test, 7) performance test, 8) system test, 9) external Beta test.

To clarify the differences between the three case studies, note that both examples are exactly the same size, but differ in these key elements:

- CMM levels
- Defect prevention
- Defect potentials
- Defect removal efficiency
- Development schedules
- Development effort
- Development costs

	A	B	C	D
1	**Testing Investment Options: ROI Analysis**			
2				
3				
4	**Testing resources**	**Case 1**	**Case 2**	**Case 3**
5		**CMM <1 Level**	**CMM 1 Level**	**CMM >3 Level**
6	Staff	$0	$60,000	$60,000
7	Infrastructure	$0	$10,000	$10,000
8	Tools	$0	$0	$12,500
9	**Total Test Investment**	**$0**	**$70,000**	**$82,500**
10				
11	**Development (Requirement, Design, Code)**			
12	Must-Fix Bugs Found	250	250	350
13	Fix Cost - $10 per bug (Internal Failure)	$2,500	$2,500	$3,500
14				
15	**Testing**			
16	Must-Fix Bugs Found	0	600	600
17	Fix Cost - $100 per bug (Internal Failure)	$0	$60,000	$60,000
18				
19	**Customer Support**			
20	Must-Fix Bugs Reported	750	150	50
21	Fix Cost - $1000 per bug (External Failure)	$750,000	$150,000	$50,000
22				
23				
24	**Cost of Quality (CoQ)**			
25	Conformance	$0	$70,000	$82,500
26	Nonconformance	$752,500	$212,500	$113,500
27	**Total CoQ**	**$752,500**	**$282,500**	**$196,000**
28				
29	**Return on Investment (ROI_1)**	**#N/A**	**571%**	**575%**
30				
31	**Return on Investment (ROI_2)**	**#N/A**	**62%**	**74%**

Fig. 11. Using Cost of Quality to Analyze two ways of Return on Investment calculation

Suppose we calculate that bugs found by testers would cost $100 to fix. This is one-tenth what a bug costs if it escapes to our customers. So, we invest $70,000 per quarterly release in a Case 2 testing process. The "**Case 2 Testing**" column shows how profitable this investment is. The testers find 600 bugs before the release, which cuts almost in 80% the number of bugs found by customers. This certainly will make the customers happier. This process improvement will also make the Chief Financial Officer happier, too: Our total cost of quality has dropped to about half a million dollars and we enjoy a nice fat 571% return on our $70,000 investment.

In some cases, we can do even better. For example, suppose that we invest $12,500 in test automation tools and Inspection activities. Let's assume we intend to recapture a return on that investment across the next twelve quarterly releases. Would we be happy if that investment in test automation helped us find about 67% more bugs?

Finding 350 bugs in development phases and 600 bugs in the test process would lower the overall customer bug find count for each release to 50. Deployment of more formal and rigorous STP in which 950 bugs out of 1000 were removed, ie. Total DRE 95%. Certainly, customers would be much happier to have the more-thoroughly tested system. In addition, cost of quality would fall to a little under $200,000, a 575% return on investment (ROI).

4.2.2 Schedule Benefits

If software quality actions are taken to reduce development cost, then this will also lead to a reduction in development schedule. We can easily calculate the reductions in the development schedule as a consequence of reductions in overall effort. In this section we will outline the schedule benefits of quality improvements.

To do so we will use the schedule estimation model from COCOMO [48].

It is instructive to understand the relationship between project size and schedule as expressed in the COCOMO II model. This is illustrated in Fig. 12. Here we see economies of scale for project schedule. This means that as the project size increases, the schedule does not increase as fast. The three lines indicate the schedule for projects employing different levels of practices. The lower risk and good practice projects tend to have a lower schedule.

Another way to formulate the ROI model in Eqn. 1 which will prove to be handy is:

$$ROI_2 = \frac{Original \cdot Total \cdot CoQ - New \cdot Total \cdot CoQ}{Original \cdot Total \cdot CoQ} \quad (2)$$

The *New Total CoQ* is defined as the total cost of software quality the project delivered after implementing the quality improvement practices or tools as in our *Case 2* and *Case 3*. This includes the cost of the investment itself. Let us look at some examples. For *Case 2* we have:

$$ROI_2 = \frac{\$752{,}500 - \$282{,}500}{\$752{,}500} = 0.62 = 62\%$$

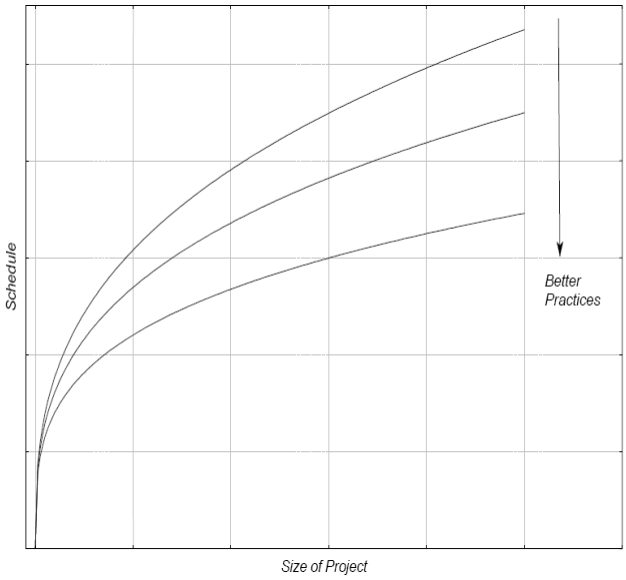

Fig. 12. Relationship between project size and schedule in COCOMO II.

This means that in *Case 2* project, the investment only saved 62% of overall project cost.

Now for *Case 3* we have:

$$ROI_2 = \frac{\$752{,}500 - \$196{,}000}{\$752{,}500} = 0.74 = 74\%, \text{ ie.}$$

the same investment saved 74% of overall project cost.

We can then formulate the New Cost as follows:

$$New\,Cost = Original\,Cost \times (1 - ROI_2)$$

Now, we can formulate the schedule reduction (SCEDRED) as a fraction (or percentage) of the original schedule as follows:

$$SCEDRED = \frac{Original \cdot Schedule - New \cdot Schedule}{Original \cdot Schedule} \quad (3)$$

By substituting the COCOMO equation for schedule, we now have:

$$SCEDRED = \frac{PM_{Original}^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)} - PM_{New}^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)}}{PM_{Original}^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)}} \quad (4)$$

where:

$PM_{Original}$ The original effort for the project in person-months

PM_{New} The new effort for the project (after implementing quality practices) in person-months

SF_j A series of five Scale Factors that are used to adjust the schedule (precedentedness, development flexibility, architecture / risk resolution, team cohesion, and process maturity).

Now, by making appropriate substitutions, we have:

$$SCEDRED = \frac{PM_{Original}^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)} - \left[PM_{Original}^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)} \times (1-ROI_2) \right]^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)}}{PM_{Original}^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)}}$$

Which simplifies to:

$$SCEDRED = 1-(1-ROI_2)^{0.28+(0.02\times\sum_{j=1}^{5} SF_j)} \quad (5)$$

The relationship between cost savings and schedule reduction is shown in Fig. 13. As can be seen, the schedule benefits tend to be at smaller proportions than the cost benefits. Nevertheless, shaving off 10% or even 5% of your schedule can have nontrivial consequences on customer relationships and market positioning.

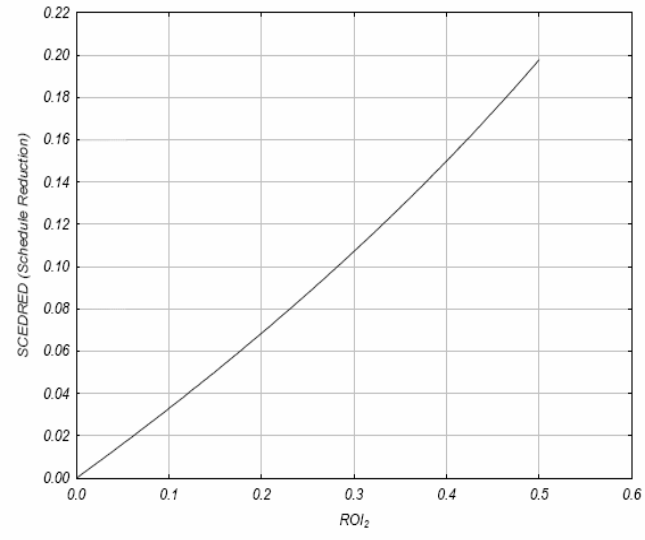

Fig. 13. The relationship between cost savings and schedule reduction for up to 50% cost savings. The assumption made for plotting this graph was that all Scale Factors were at their nominal values.

4.2.3 Interpreting the ROI Values

In this section we will explain how to interpret and use the ROI values that are calculated. First, it must be recognized that the ROI calculations, cost savings, and project costs as presented in our models are estimates. Inevitably, there is some uncertainty in these estimates. The uncertainty stems from the variables that are not accounted for in the models (there are many other factors that influence project costs, but it is not possible to account for all of these since the model would then be unusable). Another source of uncertainty is the input values themselves. These values are typically averages calculated from historical data; to the extent that the future differs from the past these values will have some error. Second, note that the calculated ROI values are for a single project. A software organization will have multiple on-going and new projects. The total benefit of implementing software quality practices to the organization can be calculated by generalizing the results to the organization. For example, if the ROI for a single project is say a 15% saving. Assuming that the input values are the same for other projects in the organization, then we can generalize to the whole organization and estimate that if software quality practices are implemented on all projects in the organization, the overall savings would be 15%. If the software budget for all the projects is say 20 million, then that would translate into an estimated saving of 3 million. Note that this is not an annual saving, but a saving in total project budgets hat may span multiple years (i.e., for the duration of the projects). To annualize it then the 15% savings must be allocated across multiple years. If you are implementing quality improvement on a single project, then these costs would have to be deducted from the single project savings. If you are implementing quality practices in the whole organization, then these costs will be spread across multiple projects. In such a case, these costs would be deducted from the organizational savings (the calculation of which is described above).

Based on the survey findings and the literature reviews, the study found the causes of difficulty that companies might experience in implementing a quality cost program. These causes can be divided into four main issues: measurement, people, process and information. Figure 14 presents the difficulty of implementation quality costs program; the effect of this diagram is an unsuccessful program.

This is a huge improvement over the initial situation. We are realizing a quantifiable and substantial return on our testing investment. We are also making our customers happier.

Management support can help in making decisions, creating a positive company environment, and providing appropriate tools and resources. These

factors interact with each other and can be explained in greater details.

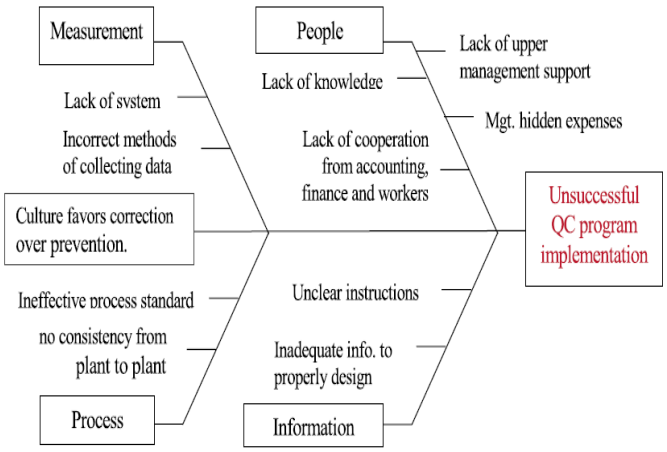

Fig. 14. Difficulty of Implementation Quality Costs Program - Cause and Effect Diagram

• *Management Support and Commitment.* Upper management and executives must drive for cost saving and understand the impact of quality costs implementation. The roles of top managers are to establish an organizational culture that favors prevention over correction, organize quality cost steering committees, meet monthly or quarterly to discuss the work progress of the quality costs program, and provide opportunity for training and learning the costs of quality for involved depart-mental managers and supervisors.

• *Effective Systems and Application.* The tools used in data collection and analysis are very important in order to obtain accurate and complete information. Each organization has a different structure; hence, quality professionals must set up the quality cost system and methodology that fit their own needs and work well with the financial and accounting systems. An effective system should be user friendly and integrated with cost drivers and collect costs related to incurred (hidden) costs. Commercial software and training for quality cost programs from the Juran Institute and the American Society for Quality are available in today's industry.

• *Understanding Concepts of Cost of Quality*. A group of respondents indicated that lack of knowledge of cost of quality caused unsuccessful quality cost implementation. It is important for everyone involved with the programs to understand the concept and elements of quality costs. There are a number of current quality cost techniques used in today's manufacturing industry, such as the quality cost model (prevention, appraisal, and failure costs), the Activity Based Costs (ABC) model, Taguchi Loss Function, Total Cost Management (process analysis and ABC), and others. These techniques might have different methods, but they all focus on the foundations of learning and training.

• *Cooperation from other departments*. Most respondents discussed the importance of cooperation from the financial and accounting departments to the quality cost program. Department managers should understand and accept the value of looking at information and acting with positive steps toward improvement. Moreover, employee involvement is also a vital issue. If workers have high job satisfaction and value preventive actions, the products will meet customers' demand as well as decrease IFC and EFC.

5 Conclusion

This paper showed that as new domains evolve and are understood there is a need to review our interpretation of quality in those new domains and where appropriate new domain-specific quality factors identified as in a new, fast growing area of Web design. CoQ is a proven technique in manufacturing industries both for communicating the value of quality initiatives and for indicating quality initiative candidates. CoSQ offers the same promise for the software industry, but has seen little use to date. CoSQ is a technique that is most useful in enabling our understanding of the economic tradeoffs involved in delivering good quality software. If software quality actions are taken to reduce development cost, then this will also lead to a reduction in development schedule. We can easily calculate the reductions in the development schedule as a consequence of reductions in overall effort as we demonstrated in one hypothetical case study.

We described Software Quality Optimization™ (SQO™) strategy as a forward-thinking approach to software quality that integrates people, processes and technologies toward one specific goal: it ensures that software deployment is synchronized with business goals to achieve competitive advantage. SQO is a continuous, iterative process throughout the application lifecycle resulting in zero-defect software that delivers value from the moment it goes live.

In this paper, we:

• Explored the true costs of software defects and their impact on application performance

• Challenged the traditional philosophy that "testing equals quality," and demonstrated how quality processes implemented throughout the application lifecycle can result in measurable performance improvements

• Shared seven best practices for optimized application quality and identified steps to implement optimized software quality processes

• Provided a definition for a quality optimization platform that drives quality efficiencies across the enterprise, and show you how investing in such a platform can help your organization minimize the costs associated with application development and realize the full potential from your application investments [12].

When these measures were introduced into large corporations such as IBM and ITT, in less than four years the volumes of delivered defects had declined by more than 50 percent, maintenance costs were reduced by more than 40 percent, and development schedules were shortened by more than 15 percent. There are no other measurements that can yield such positive benefits in such a short time span. Both customer satisfaction and employee morale improved, too, as a direct result of the reduction in defect potentials and the increase in defect removal efficiency levels.

References:

[1] Software Quality Matters, http://www.utexas.edu/coe/sqi.

[2] O. Maryoly, M. Perez, T. Rojas, "Construction of a Systemic Quality Model for Evaluating a Software Product", Kluwer Academic Publishers. Software Quality Journal, 11, 219–242, 2003.

[3] Knox, Stephen T. “Modeling the Cost of Software Quality,” Digital Technical Journal, 5:4, 9-16, 1993.

[4] D. Houston, and B. Keats, "Cost of Software Quality: A Means of Promoting Software Process Improvement", Quality Engineering, 10:3, pp. 563-573, March, 1998.

[5] R. Black, Managing the Testing Process, Second Edition. Wiley, New York, 2002.

[6] J. Campanella, ed., Principles of Quality Costs. ASQ Quality Press, Milwaukee, 1999.

[7] S. H. Kan. Metrics and Models in Software Quality Engineering, Second Edition, Addison-Wesley, 2003.

[8] Pressman, Roger, Software Engineering: A Practitioner’s Approach 3rd ed., New York: McGraw-Hill, 1992.

[9] Lj. Lazić, N. Mastorakis. “Cost Effective Software Test Metrics”, WSEAS TRANSACTIONS on COMPUTERS , Issue 6, Volume 7, June 2008, p599-619.

[10] Lj. Lazić, “MANAGING SOFTWARE QUALITY WITH DEFECTS”, 13. Telekomunikacioni Forum TELFOR2005, Beograd, 23-25 November, 2005.

[11] J. Capers. Estimating Software Costs. 2nd edition. McGraw-Hill, New York: 2007.

[12] Lj. Lazić, N. Mastorakis. “Orthogonal Array application for optimal combination of software defect detection techniques choices”, WSEAS TRANSACTIONS on COMPUTERS, Issue 8, Volume 7, August 2008, p1319-1336.

[13] M. Azuma “SQuaRE: the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality”. In ESCOM (European Software Control and Metrics conference), April 2001.

[14] M. F. Bertoa and A. Vallecillo. “Quality Attributes for COTS Components”. In Proc. of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Málaga, Spain, June 2002.

[15] R. Simao and A. Belchior. “Quality Characteristics for Software Components: Hierarchy and Quality Guides”. In LNCS 2693, pp. 188—211, Springer-Verlag, June 2003.

[16] J. Martín-Albo, M. F. Bertoa, C. Calero, A. Vallecillo, A. Cechich, M.Piattini. “CQM: A Software Component Metric Classification Model”.In Proc. of the 7th ECOOP Workshop on Quantitative Approaches inObject-Oriented Software Engineering (QAOOSE 2003). Darmstadt,Germany, June 2003.

[17] O. Preiss, A. Wegmann, and J. Wong. “On Quality Attribute BasedSoftware Engineering”, In Proc. of the 27th Euromicro Conference, Warsaw, Poland, IEEE CS Press, Sep. 2001.

[18] N.E. Fenton, S.L.O. Pfleeger, “Software Metrics: A Rigorous and Practical Approach”, published by International Thomson Computer Press, 1996.

[19] M. R. Barbacci, M. H. Klein, T. Longstaff, C. Weinstock, “Quality Attributes”, Technical Report CMU/SEI-95-TR-021, SEI CMU, Pittsburgh, 1995.

[20] B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod, M.J. Merritt, “Characteristics of Software Quality”, North Holland Publishing Company, 1978.

[21] J.A.McCall, P.K.Richards, G.F.Walters, “Factors in Software Quality”, RADC TR-77-369, Vols I, II, III, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977.

[22] ISO/IEC 9126-1:2001: “Software Engineering—Product Quality—Part1: Quality model”, June 2001.

[23] R.G. Dromey, “A Model for Software Product Quality”, IEEE Transactions on Software Engineering, 21:146-162, 1995.
[24] Robert B. Grady and Deborah L. Caswell, “Software Metrics:Establishing a Company-Wide Program”, Prentice-Hall, 1987.
[25] L.E. Hyatt, L.H.Rosenberg, “A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality”,European Space Agency Software Assurance Symposium and the 8th Annual Software Technology Conference, 1996.
[26] .D. Arthur, R.E. Nance, “A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies”, Proceedings of the 15th Annual Software Engineering Workshop, Greenbelt, Md, 1990.
[27] V.R. Basili, “Software modeling and measurement. The Goal-Question-Metric paradigm”, Computer Science Technical Report Series NR: UMIACS-TR-92-96, 1992.
[28] IEEE Standard for Software Quality Metrics Methodology, 1998.
[29] B.Kitchenham, S.Linkman, A.Pasquini, V. Nanni, “The SQUID approach to defining a quality model”, Software Quality Journal 6:211-233, 1997 .
[30] N.E.Fenton, P.Krause, M.Neil, “A Probabilistic Model for Software Defect Prediction”, accepted for publication IEEE Transactions Software Eng., 2001.
[31] N.E. Fenton, M. Neil, “Making Decisions: Using Bayesian Nets and MCDA”, Knowledge-Based Systems 14:307-325, 2001.
[32] Briand, J.Wuest, “Empirical Studies of Quality Models in Object-Oriented Systems”, Advances in Computers, 56, 2002.
[33] N. Ohlsson, H. Alberg, “Predicting fault -prone software modules in telephone switches”, IEEE Transactions on Software Eng. 22 (12): 886–894, 1996.
[34] T.M. Khoshgoftaar, E.B. Allen, “Logistic regression modelling of software quality”, International Journal of Reliability, Quality and Safety Engineering 6 (4):303-317, 1999.
[35] T.M. Khoshgoftaar, E.B. Allen, “Predicting fault -prone software modules in embedded systems with classification trees”, In Proceedings: 4th IEEE International Symposium on High-Assurance Systems Engineering, 1999.
[36] S. Bouktif, B. Kégl, H. Sahraoui, “Combin ing and Adapting Software Quality Predictive Models”, 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), 2002.
[37] M. Khoshgoftaar, E,B. Allen, “Neural networks for software quality prediction”, In W. Pedrycz, J.F. Peters, editors, Computational Intelligence in Software Engineering, 16:33-63 of Advances in Fuzzy Systems - Applications and Theory. World Scientific, 1998.
[38] K. El Emam, S.Benlarbi, N. Goel, “Comparing Case-based Reasoning Classifiers for Predicting High Risk Software Components”, Journal of Systems and Software, 2001
[39] T. M. Khoshgoftaar, E.B. Allen, W.D. Jones, J.P. Hudepohl, “Data mining for predictors of software quality”, International Journal of Software Engineering and Knowledge Engineering 9 (5): 547-563, 1999.
[40] Z. Xu, T.M. Khoshgoftaar, E.B. Allen, “Application of fuzzy expert systems in assessing operational risk of software”, Inform ation and Software Technology, 2003.
[41] G. Horgan, S. Khaddaj, P. Forte, “An essential views model for software quality assurance”, from Project Control for Software Quality, Editors, R. Kusters, A. Cowderoy, F. Heemstra, E. van Veenendaal. Shaker Publishing, 1999.
[42] Campanella, J. (Ed.). Principles of quality costs: Principles, imple¬mentation and use. (3rd ed.). New Delhi, India: Prentice-Hall, 2003.
[43] Cartin, T. J. Principles and practices of organizational perfor¬mance excellence. WI: ASQ Quality Press, 1999.
[44] Gryna, F. M. Quality and Costs. In J. M. Juran and A. B. Godfrey (Eds.), Juran’s Quality Handbook (5th ed.). (pp. 8.1-8.26). McGraw-Hill, 1999.
[45] G. Ben-Yaacov, P. Suratkar, M. Holliday & K. Bartleson, “Advancing Quality of EDA Software", Invited paper, IEEE ISQED 2002 Symposium on Quality Electronic Design, San Jose, Ca., March 18-20, 2002.
[46] S. McConnell. Professional Software Development, Addison Wesley, 2004, ISBN 0-321-19367-9
[47] B. Boehm. Software Engineering Economics; Prentice Hall, Englewood Cliffs, NJ; 1981.
[48] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer and B. Steece Software Cost Estimation with COCOMO II, Prentice Hall, 2000.
[49] National Institute of Standards & Technology, US Dept of Commerce, “The Economic Impacts of Inadequate Infrastructure for Software Testing”, May 2002.