Software Quality Management

Author: Dr. Christof Ebert
Affiliation: Vector Consulting Services GmbH, Ingersheimer Straße 24, D-70499 Stuttgart, Germany

Contact: Christof.Ebert@vector.com, www.vector.com/consulting

Abstract:
It is difficult to imagine our world without software. Since software is so ubiquitous, we need to stay in control. We have to make sure that the systems and their software run as we intend – or better. Only if software has the right quality, we will stay in control and not suddenly realize that things are going awfully wrong. Software quality management is the discipline that ensures that the software we are using and depending upon is of right quality. With solid understanding and discipline in software quality management, we can be sure that our products will deliver according to expectations, both in terms of customer and market commitments as well as business performance. This article provides a brief overview on concepts and application of software quality management. It is driven by practical experiences from many companies and is illustrated with best practices so that readers can transfer these concepts to their own environments.

Keywords:
Quality management, customer satisfaction, software, CMMI, SPICE, ISO 9001, reliability, cost of quality, cost of non-quality, defect detection, defect prediction, knowledge-base, quality control, quality assurance, verification, validation, test, review.

Acknowledgements
Some parts of sections III, IV, V, VI and VII appeared first in Ebert, C., and R.Dumke: Software Measurement. Copyrights: Springer, Heidelberg, New York, 2007. Used with permission. We recommend reading that book as an extension of the quantitative concepts mentioned in this article.

Author Biography:
Dr. Christof Ebert is managing director at Vector Consulting Services. He supports clients around the world to improve product strategy and product development and to manage organizational changes. He sits on a number of advisory and industry bodies and teaches at the University of Stuttgart.
Contact him at christof.ebert@vector.com

SOFTWARE QUALITY MANAGEMENT

Christof Ebert
Vector Consulting Services GmbH
Ingersheimer Straße 24, D-70499 Stuttgart, Germany
E-mail: Christof.Ebert@vector.com

Abstract: It is difficult to imagine our world without software. Since software is so ubiquitous, we need to stay in control. We have to make sure that the systems and their software run as we intend – or better. Only if software has the right quality, we will stay in control and not suddenly realize that things are going awfully wrong. Software quality management is the discipline that ensures that the software we are using and depending upon is of right quality. With solid understanding and discipline in software quality management, we can be sure that our products will deliver according to expectations, both in terms of customer and market commitments as well as business performance. This article provides a brief overview on concepts and application of software quality management. It is driven by practical experiences from many companies and is illustrated with best practices so that readers can transfer these concepts to their own environments.

Keywords: Quality management, customer satisfaction, software, CMMI, SPICE, ISO 9001, reliability, cost of quality, cost of non-quality, defect detection, defect prediction, knowledge-base, quality control, quality assurance, verification, validation, test, review.

I. INTRODUCTION

It is not that it has been tried and has failed,
it is that it has been found difficult and not tried.
- Gilbert Keith Chesterton

Computers and software are ubiquitous. Mostly they are embedded and we don't even realize where and how we depend on software. We might accept it or not, but software is governing our world and society and will continue further on. It is difficult to imagine our world without software. There would be no running water, food supplies, business or transportation would disrupt immediately, diseases would spread, and security would be dramatically reduced – in short, our society would disintegrate rapidly. A key reason our planet can bear over six billion people is software. Since software is so ubiquitous, we need to stay in control. We have to make sure that the systems and their software run as

we intend – or better. Only if software has the right quality, we will stay in control and not suddenly realize that things are going awfully wrong. Software quality management is the discipline that ensures that the software we are using and depending upon is of right quality. Only with solid understanding and discipline in software quality management, we will effectively stay in control.

What exactly is software quality management? To address this question we first need to define the term "quality". **Quality is the ability of a set of inherent characteristics of a product, service, product component, or process to fulfill requirements of customers** [1]. From a management and controlling perspective quality is the degree to which a set of inherent characteristics fulfills requirements. **Quality management is the sum of all planned systematic activities and processes for creating, controlling and assuring quality** [1]**.** Fig. 1 indicates how quality management relates to the typical product development. We have used a V-type visualization of the development process to illustrate that different quality control techniques are applied to each level of abstraction from requirements engineering to implementation. Quality control questions are mentioned on the right side. They are addressed by techniques such as reviews or testing. Quality assurance questions are mentioned in the middle. They are addressed by audits or sample checks. Quality improvement questions are mentioned on the left side. They are addressed by dedicated improvement projects and continuous improvement activities.

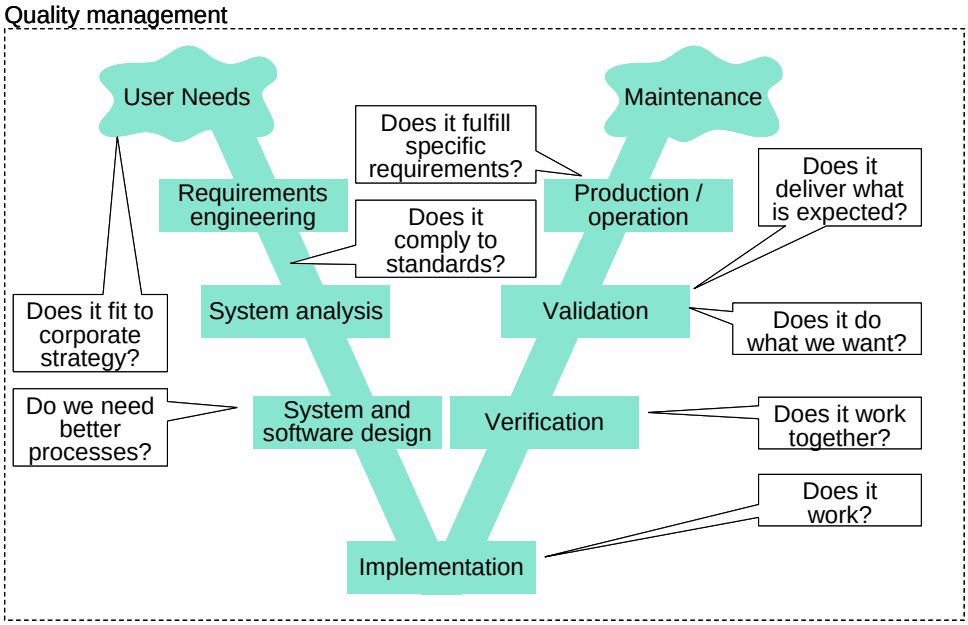

Fig. 1: Quality management techniques

Software quality management applies these principles and best practices to the development, evolution and service of software. Software quality management applies to products (e.g., assuring that it meets reliability requirements), processes (e.g., implementing defect detection techniques), projects (e.g., designing for quality) and people (e.g., evolving quality engineering skills).

Software quality is a difficult topic, because there is no defined software quality standard with objec-

tives and methods that apply to each product. Software quality has many facets, such as reliability, maintainability, security, safety, etc. Even cost can be considered a quality attribute – and it is often misinterpreted when products are designed to be of low cost and after they are released to the market suddenly have high after-sales cost. It is also difficult because there is no absolute level of quality! Quality attributes might contradict each other, such as increasing the security level by encryption and related techniques almost always negatively impacts efficiency and speed. Quality has economic impacts and depends on the corporate strategy. It is not helpful to simply state “we are a quality company” or “we need perfectionism” if there is nobody to pay for it. Quality therefore must be considered always as a topic inherent to the entire life-cycle of a product. Quality should be considered and specified during product concept and strategy activities, it must be carefully designed into the product, then it needs to be measured and verified by a multitude of activities along the life-cycle, and finally it has to be assured as long as the product and its derivatives are in use.

Software quality management is not at a standstill. With the speed at which software engineering is evolving, quality management has to keep pace. While the underlying theory and basic principles of quality management, such as quality assurance or quality control, remain invariant in the true sense (after all, they are not specific to software engineering), the application of quality management to specific contexts and situations is continuously extended. Many new application areas and methods have emerged in the recent years, such as defect prediction techniques or Six Sigma concepts. Standards have emerged which we use and explain in their practical usage. We will introduce in this article to product and service quality standards such as ISO 9001 and improvement frameworks such as the CMMI. It underlines that software quality management applies to such various domains as engineering and maintenance of IT-systems, application software, mobile computing, Web design, and embedded systems. The article is pragmatic by nature and will help you as a reader to apply concepts to your own environment. It will detail major concepts such as defect prediction, detection, correction and prevention. And it will explain relationships between quality management activities and engineering processes.

In this article we use several abbreviations and terms that might need some explanation. Generally we will use the term “software” to refer to all types of software-driven products, such as IT systems, IT services, software applications, embedded software and even firmware. The term “system” is used in a generic sense to refer to all software components of the product being developed. These components include operating systems, databases, or embedded control elements, however, they do not include hardware (which would show different reliability behaviors), as we emphasize on software products in this article. CMMI is the Capability Maturity Model Integration; ROI is return on investment; KStmt is thousand delivered executable statements of code (we use statements instead of lines, because statements are naturally the smallest unit designers deal with conceptually); PY is person-year and PH is person-hour. A failure is the deviation of system operation from requirements, for instance, the non-availability of a mobile phone connection. A defect is the underlying reason in the software that causes

the failure when it is executed, for instance, the wrong populating of a database. The concept of a defect is developer-oriented. Reliability is the probability of failure-free execution of a program for a specified period, use and environment. We distinguish between execution time which is the actual time that the system is executing the programs, and calendar time which is the time such a system is in service. A small number of defects that occur in software that is heavily used can cause a large number of failures and thus great user dissatisfaction. The number of defects over time or remaining defects is therefore not a good indicator of reliability.

II. QUALITY CONCEPTS

The long-term profitability of a company is heavily impacted by the quality perceived by customers. Customers view achieving the right balance of reliability, market window of a product and cost as having the greatest effect on their long-term link to a company [2]. This has been long articulated, and applies in different economies and circumstances. Even in restricted competitive situations, such as a market with few dominant players (e.g., the operating system market of today or the database market of few years ago), the principle applies and has given rise to open source development. With the competitor being often only a mouse-click away, today quality has even higher relevance. This applies to Web sites as well as to commodity goods with either embedded or dedicated software deliveries. And the principle certainly applies to investment goods, where suppliers are evaluated by a long list of different quality attributes.

Methodological approaches to guarantee quality products have lead to international guidelines (e.g., ISO 9001 [3]) and widely applied methods to assess the development processes of software providers (e.g., SEI CMMI [4,5]). In addition, most companies apply certain techniques of criticality prediction that focus on identifying and reducing release risks [6,7,8]. Unfortunately, many efforts usually concentrate on testing and reworking instead of proactive quality management [9].

Yet there is a problem with quality in the software industry. By quality we mean the bigger picture, such as delivering according to commitments. While solutions abound, knowing which solutions work is the big question. What are the most fundamental underlying principles in successful projects? What can be done right now? What actually is good or better? What is good enough – considering the immense market pressure and competition across the globe?

A simple – yet difficult to digest and implement – answer to these questions is that software quality management is not simply a task, but rather a habit. It must be engrained in the company culture. It is something that is visible in the way people are working, independent on their role. It certainly means that every single person in the organization sees quality as her own business, not that of a quality manager or a testing team. A simple yet effective test to quickly identify the state of practice with respect to quality management is to ask around what quality means for an employee and how he delivers according to this meaning. You will identify that many see it as a bulky and formal approach to be done

to achieve necessary certificates. Few exceptions exist, such as industries with safety and health impacts. But even there, you will find different approaches to quality, depending on culture. Those with carrot and stick will not achieve a true quality culture. Quality is a habit. It is driven by objectives and not based on believes. It is primarily achieved when each person in the organization knows and is aware on her own role to deliver quality.

Quality management is the responsibility of the entire enterprise. It is strategically defined, directed and operationally implemented on various organizational levels. Fig. 2 shows in a simplified organizational layout with four tiers the respective responsibilities to successfully implement quality management. Note that it is not a top-down approach where management sets unrealistic targets that must be implemented on the project level. It is even more important that continuous feedback is provided bottom-up so that decisions can be changed or directions can be adjusted.

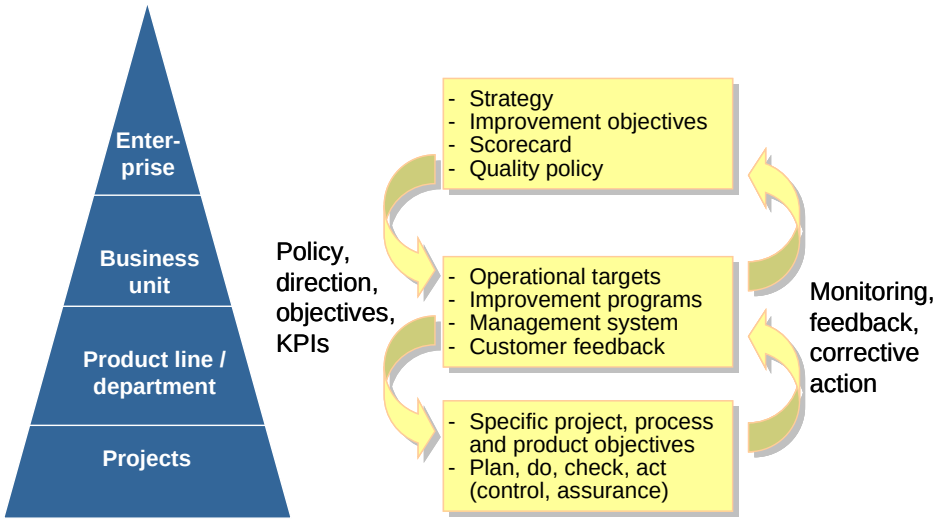

Fig. 2: Quality management within the organization

Quality is implemented along the product life-cycle. Fig. 3 shows some pivotal quality-related activities mapped to the major life-cycle phases. Note that on the left side strategic directions are set and a respective management system is implemented. Towards the right side, quality related processes, such as test or supplier audits are implemented. During evolution of the product with dedicated services and customer feedback, the product is further optimized and the management system is adapted where necessary.

It is important to understand that the management system is not specific to a project, but drives a multitude of projects or even the entire organization. Scale effects occur with having standardized processes that are systematically applied. This not only allows moving people to different projects without long learning curve but also assures proven quality at best possible efficiency. A key step along all these phases is to recognize that all quality requirements can and should be specified in quantitative terms. This does not mean “counting defects” as it would be too late and reactive. It means quantifying quality attributes such as security, portability, adaptability, maintainability, robustness, usability, relia-

bility and performance as an objective *before* the design of a product [1].

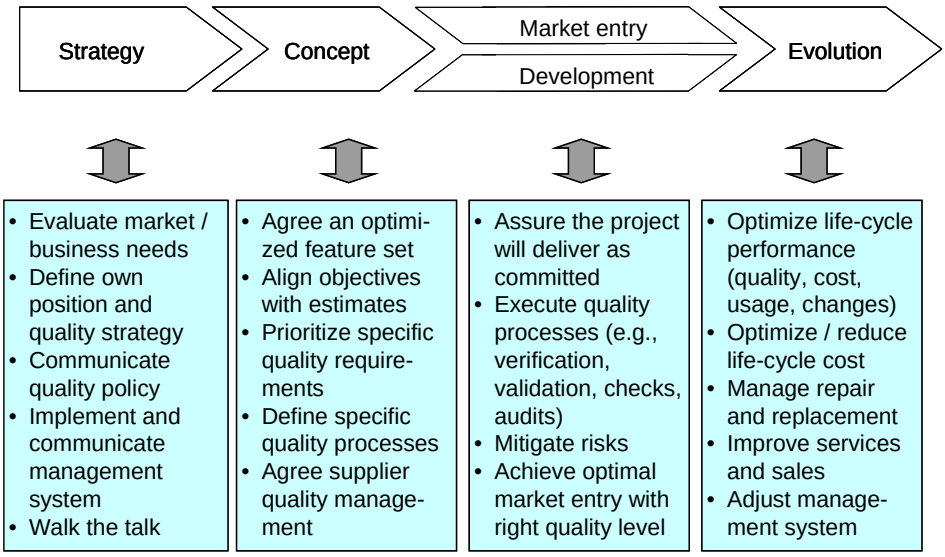

Fig. 3: Quality management along the product life-cycle

A small example will illustrate this need. A software system might have strict reliability constraints. Instead of simply stating that reliability should achieve less than one failure per month in operation, which would be reactive, related quality requirements should target underlying product and process needs to achieve such reliability. During the strategy phase, the market or customer needs for reliability need to be elicited. Is reliability important as an image or is it rather availability? What is the perceived value of different failure rates? A next step is to determine how these needs will be broken down to product features, components and capabilities. Which architecture will deliver the desired reliability and what are cost impacts? What component and supplier qualification criteria need to be established and maintained throughout the product life-cycle? Then the underlying quality processes need to be determined. This should not be done ad-hoc and for each single project individually but by tailoring organizational processes, such as product life-cycle, project reviews, or testing to the specific needs of the product. What test coverage is necessary and how will it be achieved? Which test equipment and infrastructure for interoperability of components needs to be applied? What checklists should be used in preparing for reviews and releases? These processes need to be carefully applied during development. Quality control will be applied by each single engineer and quality assurance will be done systematically for selected processes and work products. Finally the evolution phase of the product needs to establish criteria for service request management and assuring the right quality level of follow-on releases and potential defect corrections. A key question to address across all these phases is how to balance quality needs with necessary effort and availability of skilled people. Both relates to business, but that is at times overlooked. We have seen companies that due to cost and time constraints would reduce requirements engineering or early reviews of specifications and later found out that follow-on cost were higher than what was cut out. A key understanding to achieving quality and

therefore business performance has once been phrased by Abraham Lincoln: “If someone gave me eight hours to chop down a tree, I would spend six hours sharpening the axe.”

III. PROCESS MATURITY AND QUALITY

The quality of a product or service is mostly determined by the processes and people developing and delivering the product or service. Technology can be bought, it can be created almost on the spot, and it can be introduced by having good engineers. What matters to the quality of a product is how they are working and how they introduce this new technology. Quality is not at a stand-still, it needs to be continuously questioned and improved. With today’s low entry barriers to software markets, one thing is sure: There is always a company or entrepreneur just approaching your home turf and conquering your customers. To continuously improve and thus stay ahead of competition, organizations need to change in a deterministic and results-oriented way. They need to know and improve their process maturity.

The concept of process maturity is not new. Many of the established quality models in manufacturing use the same concept. This was summarized by Philip Crosby in his bestselling book “Quality is Free” in 1979 [10]. He found from his broad experiences as a senior manager in different industries that business success depends on quality. With practical insight and many concrete case studies he could empirically link process performance to quality. His credo was stated as: “Quality is measured by the cost of quality which is the expense of nonconformance – the cost of doing things wrong.”

First organizations must know where they are, they need to assess their processes. The more detailed the results from such an assessment, the easier and more straightforward it is to establish a solid improvement plan. That was the basic idea with the “maturity path” concept proposed by Crosby in the 1970s. He distinguishes five maturity stages, namely

- Stage 1: Uncertainty
- Stage 2: Awakening
- Stage 3: Enlightening
- Stage 4: Wisdom
- Stage 5: Certainty

These five stages were linked to process measurements. Crosby looked into the following categories of process measurement: Management understanding and attitude, quality organization status, problem handling, cost of quality as percentage of sales, quality improvement actions with progress and overall organizational quality posture.

The concept of five maturity stages further evolved in the software engineering domain under the lead of Watts Humphrey and his colleagues in the 1980s to eventually build the Capability Maturity Model [11]. Effective process improvement can be achieved by following the widely-used **Capability Maturity Model Integration (CMMI)**, originally issued by the Software Engineering Institute [4,5].

Based on structuring elements of maturity levels, goals and practices, the CMMI offers a well-defined appraisal technique to assess your own or your suppliers' processes and to benchmark performance. It extends the "classical" **ISO 9001** to an improvement view integrating the systems engineering, acquisition and all engineering processes along the product life-cycle. **ISO 15504** is the world-wide standard for improvement frameworks and supplier assessments [12]. It has been derived from experiences with the Capability Maturity Model during the nineties. A similar but less used process improvement framework is the **SPICE** model. SPICE stands for "software process improvement and capability determination". Both frameworks, namely CMMI and SPICE, are fully compatible and based on ISO 15504, which governs capability assessments on a global scale.

Let us look more closely to the CMMI as it has the widest usage in software and IT industries worldwide to implement quality processes. The CMMI provides a framework for process improvement and is used by many software development organizations. It defines five levels of process maturity plus an underlying improvement framework for process maturity and as a consequence, quality and performance. Table 1 shows the five maturity levels of the CMMI and what they imply in terms of engineering and management culture. It is structured by so-called process areas which can be seen as containers with goals a process must meet in order to deliver good results and best industry practices that can be used to achieve these process goals.

Table 1: The five maturity levels of the CMMI and their respective impact on performance

CMMI Maturity Level	Title	What it means
5	Optimizing	Continuous process improvement on all levels. Business objectives closely linked to processes. Deterministic change management.
4	Managed	Quantitatively predictable product and process quality. Well-managed, business needs drive results.
3	Defined	Standardized and tailored engineering and management process. Predictable results, good quality, focused improvements, reduced volatility of project results, objectives are reached.
2	Repeatable	Project management and commitment process. Increasingly predictable results, quality improvements.
1	Initial	Ad-hoc, chaotic, poor quality, delays, missed commitments.

As an example we can take the "Requirements Management" process area which one such goal, namely, "requirements are managed and inconsistencies with project plans and work products are identified." Furtheron the CMMI provides so-called generic practices that equally apply to each process area and drive institutionalization and culture change. There are five such generic goals with one allocated to each level, namely (1) "achieve specific goals", (2) "institutionalize a managed process", (3) "institutionalize a defined process", (4) "institutionalize a quantitatively managed process", and (5) "institutionalize an optimizing process". The first of these generic goals underlines that first the basic functional content of a process needs to be implemented. The four subsequent goals build upon each other

and show the path to maturity by first managing a process, then building on previous experiences and defining the process, then managing the process quantitatively, and finally to continuously improve the process by removing common causes of variation. In order to see the effectiveness, efficiency and performance of a single process area, measurements are applied that suit the needs of each of these five levels..

Why do software organizations embark on frameworks such as the CMMI or SPICE to improve processes and products? There are several answers to this question. Certainly it is all about competition. Companies have started to realize that momentum is critical: If you stand still, you fall behind! The business climate and the software marketplace have changed in favor of end users and customers. Companies need to fight for new business, and customers expect process excellence. A widely used and industry-proven process improvement framework such as the CMMI offers the benefit to improve on a determined path, to benchmark with other companies and to apply worldwide supplier audits or comparisons on a standardized scale.

Such process improvement frameworks can provide useful help if introduced and orchestrated well. A key success factor is to not get trapped into certification as the major goal. This was a disease towards the end of last century, where many companies claimed having quality processes just because they had certificates on the walls. Since quality was not entrenched in their culture, they continued with the same practices and ad-hoc firefighting style they had before. The only difference was that somewhere they had “the book” where lots of theoretic processes were described to pass an audit. But engineers were still driven into projects with insufficient resources and customer commitments were still made before understanding impacts of changes. As a result such companies delivered insufficient quality with highly inefficient processes and thus faced severe problems in their markets. At times such companies and their management would state that ISO 9001 or CMMI are of no value and only drive heavy processes, but as we have learned in the introductory chapters of this article, it was essentially a failure – their failure – of quality leadership and setting the right directions.

Let us see how to avoid such traps and stay focused on business. The primary answer is to follow an objective-driven improvement approach, with objectives closely connected to business needs. Using the CMMI and its process areas for an objective-driven improvement increment consists of the following steps:

1. Identify the organization’s business objectives and global improvement needs.
2. Define and agree on the organization’s key performance indicators (KPIs) based on the organization’s business objectives as the primary instrument to relate improvement and performance to individual objectives and thus drive culture change bottom-up and top-down.
3. Identify the organization’s areas of weakness or areas where they are feeling the most pain internally or from customers from e.g., root cause analysis, customer surveys, and so on.
4. Commit to concrete and measurable improvement objectives determined on a level where they are understandable and obvious how to implement.

5. Identify those CMMI Process Areas (PAs) that will best support improvements for the areas identified in Step 3 and provide a return on investment (ROI).
6. Perform an initial “gap analysis” of the PAs identified in Step 5 to identify strengths and weaknesses and to baseline initial performance measurements, such as efficiency, effectiveness, quality, cost, etc.
7. Develop a plan for those PAs that you want to focus on first and make sure that the plan is sufficiently detailed to see milestones, responsibilities, objectives and actions along the way until the improvement objectives resulting from step 4 are reached.
8. Implement change and measure progress against the committed improvement objectives from step 4.
9. Follow up results, risks and performance with senior management on a periodic basis to assure that global objectives of step 2 are achieved.

The trend in the industry as a whole is growing towards higher maturity levels. In many markets such as government, defense and automotive the clear demand is for a maturity level of 2 or 3. In supplier markets it can well be maturity levels 4 or 5 which is a strong assurance that your supplier not only has processes defined, but that he manages his processes quantitatively and continuously improves.

It is all about business: Your competitors are at least at this same place (or ahead of you). And they carefully look to their processes and continuously improve them. You might neglect it and claim that your products and market position are far from competition. Well just look towards the speed of change in IT and software markets. Be it Microsoft loosing ground to Google or the many IT entries to Fortune 500 disappearing within three to five years totally, the message is that they all thought that they have a safe position. The road to IT and software success is littered by the many skeletons of those who once thought that they are invincible and in consequence forgot to pay attention to clients and own performance. The goal thus is to continuously improve engineering processes, lower costs, increase adherence to commitments and improve product quality.

IV. DEFECTS – PREDICTION, DETECTION, CORRECTION AND PREVENTION

To achieve the right quality level in developing a product it is necessary to understand what it means not to have insufficient quality. Let us start with the concept of defects. A defect is defined as an imperfection or deficiency in a system or component where that component does not meet its requirements or specifications which could yield a failure. Causal relationship distinguishes the failure caused by a defect which itself is caused by a human error during design of the product. Defects are not just information about something wrong in a software system or about the progress in building up quality.

Defects are information about problems in the process that created this software. The four questions to address are:

1. How many defects are there and have to be removed?

2. How can the critical and relevant defects be detected most efficiently?
3. How can the critical and relevant defects be removed most effectively and efficiently?
4. How can the process be changed to avoid the defects from reoccurring?

These four questions relate to the four basic quality management techniques of **prediction, detection, correction and prevention**. The first step is to identify how many defects there are and which of those defects are critical to product performance. The underlying techniques are statistical methods of defect estimation, reliability prediction and criticality assessment. These defects have to be detected by quality control activities, such as inspections, reviews, unit test, etc. Each of these techniques has their strengths and weaknesses which explains why they ought to be combined to be most efficient. It is of not much value to spend loads of people on test, when in-depth requirements reviews would be much faster and cheaper. Once defects are detected and identified, the third step is to remove them. This sounds easier than it actually is due to the many ripple effects each correction has to a system. Regression tests and reviews of corrections are absolutely necessary to assure that quality won't degrade with changes. A final step is to embark on preventing these defects from re-occurring. Often engineers and their management state that this actually should be the first and most relevant step. We agree, but experience tells that again and again, people stumble across defect avoidance simply because their processes won't support it. In order to effectively avoid defects engineering processes must be defined, systematically applied and quantitatively managed. This being in place, defect prevention is a very cost-effective means to boost both customer satisfaction and business performance, as many high-maturity organizations such as Motorola, Boeing or Wipro show [1,13].

Defect removal is not about assigning blame but about building better quality and improving the processes to ensure quality. Reliability improvement always needs measurements on effectiveness (i.e., percentage of removed defects for a given activity) compared to efficiency (i.e., effort spent for detecting and removing a defect in the respective activity). Such measurement asks for the number of residual defects at a given point in time or within the development process.

But how is the number of defects in a piece of software or in a product estimated? We will outline the approach we follow for up-front estimation of residual defects in any software that may be merged from various sources with different degrees of stability. We distinguish between upfront defect estimation which is static by nature as it looks only on the different components of the system and their inherent quality before the start of validation activities, and reliability models which look more dynamically during validation activities at residual defects and failure rates.

Only a few studies have been published that typically relate static defect estimation to the number of already detected defects independently of the activity that resulted in defects [8,14], or the famous error seeding [15] which is well known but is rarely used due to the belief of most software engineers that it is of no use to add errors to software when there are still far too many defects in, and when it is known that defect detection costs several person hours per defect.

Defects can be easily estimated based on the stability of the underlying software. All software in a

product can be separated into four parts according to its origin:

- Software that is new or changed. This is the standard case where software had been designed especially for this project, either internally or from a supplier.
- Software reused but to be tested (i.e., reused from another project that was never integrated and therefore still contains lots of defects; this includes ported functionality). This holds for reused software with unclear quality status, such as internal libraries.
- Software reused from another project that is in testing (almost) at the same time. This software might be partially tested, and therefore the overlapping of the two test phases of the parallel projects must be accounted for to estimate remaining defects. This is a specific segment of software in product lines or any other parallel usage of the same software without having hardened it so far for field usage.
- Software completely reused from a stable product. This software is considered stable and therefore it has a rather low number of defects. This holds especially for commercial off the shelf software components and open source software which is used heavily.

The base of the calculation of new or changed software is the list of modules to be used in the complete project (i.e., the description of the entire build with all its components). A defect correction in one of these components typically results in a new version, while a modification in functionality (in the context of the new project) results in a new variant. Configuration management tools are used to distinguish the one from the other while still maintaining a single source.

To statically estimate the number of residual defects in software at the time it is delivered by the author (i.e., after the author has done all verification activities, she can execute herself), we distinguish four different levels of stability of the software that are treated independently [1]:

$$f = a \times x + b \times y + c \times z + d \times (w - x - y - z)$$

with

- x: the number of new or changed KStmt designed and to be tested within this project. This software was specifically designed for that respective project. All other parts of the software are reused with varying stability.
- y: the number of KStmt that are reused but are unstable and not yet tested (based on functionality that was designed in a previous project or release, but was never externally delivered; this includes ported functionality from other projects).
- z: the number of KStmt that are tested in parallel in another project. This software is new or changed for the other project and is entirely reused in the project under consideration.
- w: the number of KStmt in the total software – i.e., the size of this product in its totality.

The factors *a-d* relate defects in software to size. They depend heavily on the development environment, project size, maintainability degree and so on. Our starting point for this initial estimation is actually driven by psychology. Any person makes roughly one (non-editorial) defect in ten written lines

of work. This applies to code as well as a design document or e-mail, as was observed by the personal software process (PSP) and many other sources [1,16,17]. The estimation of remaining defects is language independent because defects are introduced per thinking and editing activity of the programmer, i.e., visible by written statements.

This translates into 100 defects per KStmt. Half of these defects are found by careful checking by the author which leaves some 50 defects per KStmt delivered at code completion. Training, maturity and coding tools can further reduce the number substantially. We found some 10-50 defects per KStmt depending on the maturity level of the respective organization [1]. This is based only on new or changed code, not including any code that is reused or automatically generated.

Most of these original defects are detected by the author before the respective work product is released. Depending on the underlying individual software process, 40–80% of these defects are removed by the author immediately. We have experienced in software that around 10–50 defects per KStmt remain. For the following calculation we will assume that 30 defects/KStmt are remaining (which is a common value [18]. Thus, the following factors can be used:

- *a*: 30 defects per KStmt (depending on the engineering methods; should be based on own data)
- *b*: 30 × 60% defects per KStmt, if defect detection before the start of testing is 60%
- *c*: 30 × 60% × (overlapping degree) × 25% defects per KStmt (depending on overlapping degree of resources)
- *d*: 30 × 0.1–1% defects per KStmt depending on the number of defects remaining in a product at the time when it is reused

The percentages are, of course, related to the specific defect detection distribution in one's own historical database (Fig. 4). A careful investigation of stability of reused software is necessary to better substantiate the assumed percentages.

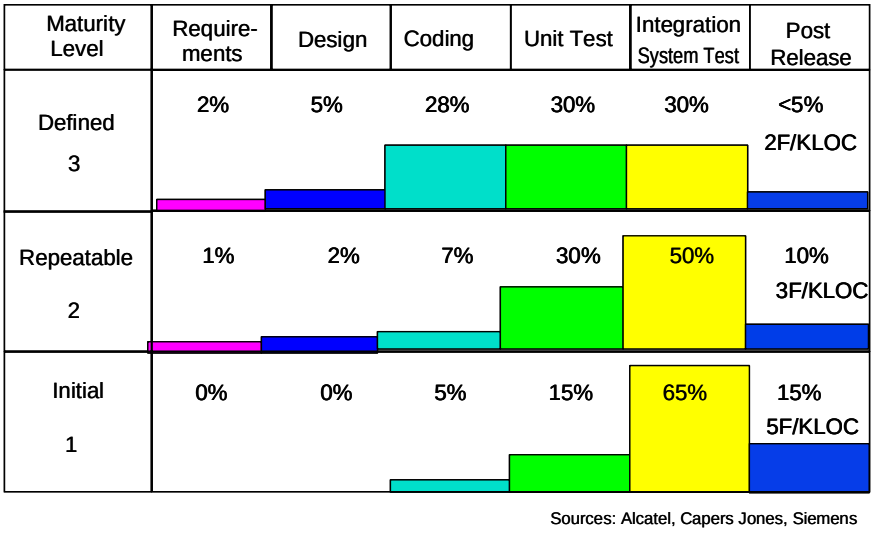

Fig. 4: Typical benchmark effects of detecting defects earlier in the life-cycle

Since defects can never be entirely avoided, different quality control techniques are used in combina-

tion for detecting defects during the product life-cycle. They are listed in sequence when they are applied throughout the development phase, starting with requirements and ending with system test:

- Requirements (specification) reviews
- Design (document) reviews
- Compile level checks with tools
- Static code analysis with automatic tools
- Manual code reviews and inspections with checklists based on typical defect situations or critical areas in the software
- Enhanced reviews and testing of critical areas (in terms of complexity, former failures, expected defect density, individual change history, customer's risk and occurrence probability)
- Unit testing
- Focused testing by tracking the effort spent for analyses, reviews, and inspections and separating according to requirements to find out areas not sufficiently covered
- Systematic testing by using test coverage measurements (e.g., C0 and C1 coverage) and improvement
- Operational testing by dynamic execution already during integration testing
- Automatic regression testing of any redelivered code
- System testing by applying operational profiles and usage specifications.

We will further focus on several selected approaches that are applied for improved defect detection before starting with integration and system test because those techniques are most cost-effective.

Note that the starting point for effectively reducing defects and improving reliability is to track all defects that are detected. Defects must be recorded for each defect detection activity. Counting defects and deriving the reliability (that is failures over time) is the most widely applied and accepted method used to determine software quality. Counting defects during the complete project helps to estimate the duration of distinct activities (e.g., unit testing or subsystem testing) and improves the underlying processes. Failures reported during system testing or field application must be traced back to their primary causes and specific defects in the design (e.g., design decisions or lack of design reviews).

Quality improvement activities must be driven by a careful look into what they mean for the bottom line of the overall product cost. It means to continuously investigate what this best level of quality really means, both for the customers and for the engineering teams who want to deliver it.

One does not build a sustainable customer relationship by delivering bad quality and ruining his reputation just to achieve a specific delivery date. And it is useless to spend an extra amount on improving quality to a level nobody wants to pay for. The optimum seemingly is in between. It means to achieve the right level of quality and to deliver in time. Most important yet is to know from the begin of the project what is actually relevant for the customer or market and set up the project accordingly. Objectives will be met if they are driven from the beginning.

We look primarily at factors such as cost of non-quality to follow through this business reasoning of

quality improvements. For this purpose we measure all cost related to error detection and removal (i.e., cost of non-quality) and normalize by the size of the product (i.e., normalize defect costs). We take a conservative approach in only considering those effects that appear inside our engineering activities, i.e., not considering opportunistic effects or any penalties for delivering insufficient quality.

The most cost-effective techniques for defect detection are requirements reviews [1]. For code reviews, inspections and unit test are most cost-effective techniques aside static code analysis. Detecting defects in architecture and design documents has considerable benefit from a cost perspective, because these defects are expensive to correct at later stages. Assuming good quality specifications, major yields in terms of reliability, however, can be attributed to better code, for the simple reason that there are many more defects residing in code that were inserted during the coding activity. We therefore provide more depth on techniques that help to improve the quality of code, namely code reviews (i.e., code reviews and formal code inspections) and unit test (which might include static and dynamic code analysis).

There are six possible paths of combining manual defect detection techniques in the delivery of a piece of software from code complete until the start of integration test (Fig. 5). The paths indicate the permutations of doing code reviews alone, performing code inspections and applying unit test. Each path indicated by the arrows shows which activities are performed on a piece of code. An arrow crossing a box means that the activity is not applied. Defect detection effectiveness of a code inspection is much higher than that of a code review [1]. Unit test finds different types of defects than reviews. However cost also varies depending on which technique is used, which explains why these different permutations are used. In our experience code reviews is the cheapest detection technique (with ca. 1-2 PH/defect) , while manual unit test is the most expensive (with ca. 1-5 PH/defect, depending on automation degree). Code inspections lie somewhere in between. Although the best approach from a mere defect detection perspective is to apply inspections and unit test, cost considerations and the objective to reduce elapsed time and thus improve throughput suggest carefully evaluating which path to follow in order to most efficiently and effectively detect and remove defects.

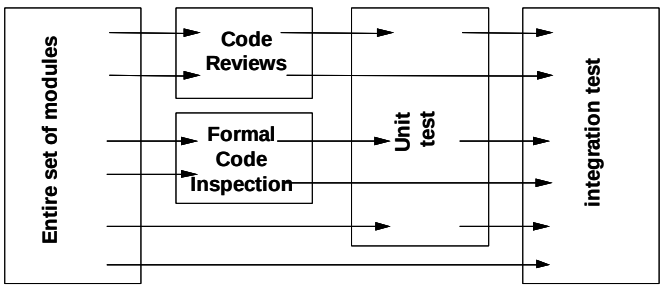

Fig. 5: Six possible paths for modules between end of coding and start of integration test

Unit tests, however, combined with C0 coverage targets, have the highest effectiveness for regression testing of existing functionality. Inspections, on the other hand, help in detecting distinct defect classes that can only be found under real load (or even stress) in the field.

Defects are not distributed homogeneously through new or changed code [1,19]. An analysis of many projects revealed the applicability of the Pareto rule: 20-30% of the modules are responsible for 70-80% of the defects of the whole project [1,16,20]. These critical components need to be identified as early as possible, i.e., in the case of legacy systems at start of detailed design, and for new software during coding. By concentrating on these components the effectiveness of code inspections and unit testing is increased and fewer defects have to be found during test phases. By concentrating on defect-prone modules both effectiveness and efficiency are improved. Our main approach to identify defect-prone software-modules is a criticality prediction taking into account several criteria. One criterion is the analysis of module complexity based on complexity measurements. Other criteria concern the number of new or changed code in a module, and the number of field defects a module had in the preceding project. Code inspections are first applied to heavily changed modules, in order to optimize payback of the additional effort that has to be spent compared to the lower effort for code reading. Formal code reviews are recommended even for very small changes with a checking time shorter than two hours in order to profit from a good efficiency of code reading. The effort for know-how transfer to another designer can be saved.

It is of great benefit for improved quality management to be able to predict early on in the development process those components of a software system that are likely to have a high defect rate or those requiring additional development effort. Criticality prediction is based on selecting a distinct small share of modules that incorporate sets of properties that would typically cause defects to be introduced during design more often than in modules that do not possess such attributes. Criticality prediction is thus a technique for risk analysis during the design process.

Criticality prediction addresses typical questions often asked in software engineering projects:

- How can I early identify the relatively small number of critical components that make significant contribution to defects identified later in the life-cycle?
- Which modules should be redesigned because their maintainability is bad and their overall criticality to the project’s success is high?
- Are there structural properties that can be measured early in the code to predict quality attributes?
- If so, what is the benefit of introducing a measurement program that investigates structural properties of software?
- Can I use the often heuristic design and test know-how on trouble identification and risk assessment to build up a knowledge base to identify critical components early in the development process?

Criticality prediction is a multifaceted approach taking into account several criteria [21]. Complexity is a key influence on quality and productivity. Having uncontrolled accidental complexity in the product will definitely decrease productivity (e.g., gold plating, additional rework, more test effort) and quality (more defects). A key to controlling accidental complexity from creeping into the project is the

measurement and analysis of complexity throughout in the life-cycle. Volume, structure, order or the connections of different objects contribute to complexity. However, do they all account for it equally? The clear answer is no, because different people with different skills assign complexity subjectively, according to their experience in the area. Certainly criticality must be predicted early in the life-cycle to effectively serve as a managerial instrument for quality improvement, quality control effort estimation and resource planning as soon as possible in a project. Tracing *comparable* complexity metrics for different products throughout the life-cycle is advisable to find out when essential complexity is overruled by accidental complexity. Care must be used that the complexity metrics are comparable, that is they should measure the same factors of complexity.

Having identified such overly critical modules, risk management must be applied. The most critical and most complex, for instance, the top 5% of the analyzed modules are candidates for a redesign. For cost reasons mitigation is not only achieved with redesign. The top 20% should have a code inspection instead of the usual code reading, and the top 80% should be at least entirely (C0 coverage of 100%) unit tested. By concentrating on these components the effectiveness of code inspections and unit test is increased and fewer defects have to be found during test phases. To achieve feedback for improving predictions the approach is integrated into the development process end-to-end (requirements, design, code, system test, deployment).

It must be emphasized that using criticality prediction techniques does not mean attempting to detect all defects. Instead, they belong to the set of managerial instruments that try to optimize resource allocation by focusing them on areas with many defects that would affect the utility of the delivered product. The trade-off of applying complexity-based predictive quality models is estimated based on

- limited resources are assigned to high-risk jobs or components
- impact analysis and risk assessment of changes is feasible based on affected or changed complexity
- gray-box testing strategies are applied to identified high-risk components
- fewer customers reported failures

Our experiences show that, in accordance with other literature [22] corrections of defects in early phases is more efficient, because the designer is still familiar with the problem and the correction delay during testing is reduced.

The effect and business case for applying complexity-based criticality prediction to a new project can be summarized based on results from our own experience database (taking a very conservative ratio of only 40% defects in critical components):

- 20% of all modules in the project were predicted as most critical (after coding);
- these modules contained over 40% of all defects (up to release time).

Knowing from these and many other projects that

- 60% of all defects can theoretically be detected until the end of unit test and

- defect correction during unit test and code reading costs less than 10% compared to defect correction during system test,

it can be calculated that 24% of all defects can be detected early by investigating 20% of all modules more intensively with 10% of effort compared to late defect correction during test, therefore yielding a 20% total cost reduction for defect correction. Additional costs for providing the statistical analysis are in the range of two person days per project. Necessary tools are off the shelf and account for even less per project.

V. RELIABILITY

Software reliability engineering is a statistical procedure associated with test and correction activities during development. It is further used after delivery, based on field data, to validate prediction models. Users of such prediction models and the resulting reliability values are development managers who apply them to determine the best suitable defect detection techniques and to find the optimum delivery time. Operations managers use the data for deciding when to include new functionality in a delivered product that is already performing in the field. Maintenance managers use the reliability figures to plan the allocation of resources they need to deliver corrections in due time according to contracted deadlines.

The current approach to software reliability modeling focuses on the testing and rework activities of the development process. On the basis of data on the time intervals between occurrences of failures, collected during testing, we attempt to make inferences about how additional testing and rework would improve the reliability of the product and about how reliable the product would be once it is released to the user.

Software reliability engineering includes the following activities [1,8,23]:

- selecting a mixture of quality factors oriented towards maximizing customer satisfaction
- determining a reliability objective (i.e., exit criteria for subsequent test and release activities)
- predicting reliability based on models of the development process and its impact on defect introduction, recognition and correction
- supporting test strategies based on realization (e.g., white box testing, control flow branch coverage) or usage (e.g., black box testing, operational profiles)
- providing insight to the development process and its influence on software reliability
- improving the software development process in order to obtain higher quality reliability
- defining and validating measurements and models for reliability prediction

The above list of activities is derived mainly from the customer's point of view. When making the distinction between failures and defects, the customer is interested in the reduction of failures. Emphasis on reducing failures means that development and testing is centered towards functions in normal and extraordinary operational modes (e.g., usage coverage instead of branch coverage during system test-

ing, or operational profiles instead of functional profiles during reliability assessment). In this section we will focus on the aspect of reliability modeling that is used for measuring and estimating (predicting) the reliability of a software release during testing as well as in the field.

A **reliability prediction model** is primarily used to help in project management (e.g., to determine release dates, service effort or test effort) and in engineering support (e.g., to provide insight on stability of the system under test, to predict residual defects, to relate defects to test cases and test effectiveness). It is also used for critical management decisions, such as evaluating the trade-off of releasing the product with a certain number of expected failures or to determine the necessary service resources after release. The reliability model is based upon a standard model that is selected by the quality engineer from the literature of such models and their applicability and scope [8]. Then it is adapted and tailored to the actual situation of test and defect correction. Detected defects are reported into the model together with underlying execution time. More sophisticated models also take into consideration the criticality of software components, such as components with a long defect history or those that had been changed several times during the current or previous releases.

Test strategies must closely reflect operational scenarios and usage in order to predict reliability after release. The model then will forecast the defects or failures to expect during test or after release. Accuracy of such models should be in the range of 20-30 percent to ensure meaningful decisions. If they are far off, the wrong model had been selected (e.g., not considering defects inserted during correction cycles) or the test strategy is not reflecting operational usage.

Such models use an appropriate statistical model which requires accurate test or field failure data related to the occurrences in terms of execution time. Execution time in reliability engineering is the accumulated time a system is executed under real usage conditions. It is used for reliability measurement and predictions to relate individual test time and defect occurrence towards the would-be performance under real usage conditions.

Several models should be considered and assessed for their predictive accuracy, in order to select the most accurate and reliable model for reliability prediction. It is of no use to switch models after the facts to achieve best fit, because then you would have no clue about how accurate the model would be in a predictive scenario. Unlike many research papers on that subject, our main interest is to select a model that would provide in very different settings (i.e., project sizes) of the type of software we are developing a good fit that can be used for project management. Reliability prediction thus should be performed at intermediate points and at the end of system testing.

At intermediate points, reliability predictions will provide a measurement of the product’s current reliability and its growth, and thus serve as an instrument for estimating the time still required for test. They also help in assessing the trade-off between extensive testing and potential loss of market share (or penalties in case of investment goods) because of late delivery. At the end of development which is the decision review before releasing the product to the customer, reliability estimations facilitate an evaluation of reliability against the committed targets. Especially in communication, banking and de-

fense businesses, such reliability targets are often contracted and therefore are a very concrete exit criterion. It is common to have multiple failure rate objectives.

For instance, failure rate objectives will generally be lower (more stringent) for high failure severity classes. The factors involved in setting failure rate objectives comprise the market and competitive situation, user satisfaction targets and risks related to defects of the system. Life-cycle costs related to development, test and deployment in a competitive situation must be carefully evaluated, to avoid setting reliability objectives too high.

Reliability models are worthless if they are not continuously validated against the actually observed failure rates. We thus also include in our models which will be presented later in this section, a plot of predicted against actually observed data.

The application of reliability models to reliability prediction is based on a generic algorithm for model selection and tuning:

1. Establish goals according to the most critical process areas or products (e.g., by following the quality improvement paradigm or by applying a Pareto analysis).
2. Identify the appropriate data for analyzing defects and failures (i.e., with classifications according to severity, time between failures, reasons for defects, test cases that helped in detection and so on).
3. Collect data relevant for models that help to gain insight into how to achieve the identified goals. Data collection is cumbersome and exhaustive: Tools may change, processes change as well, development staff are often unwilling to provide additional effort for data collection and – worst of all – management often does not wish for changes that affect it personally.
4. Recover historical data that was available at given time stamps in the software development process (for example, defect rates of testing phases); the latter of these will serve as starting points of the predictive models.
5. Model the development process and select a defect introduction model, a testing model and a correction model that suit the observed processes best.
6. Select a reliability prediction model that suits the given defect introduction, testing and correction models best.
7. Estimate the parameters for the reliability model using only data that were available at the original time stamps.
8. Extrapolate the function at some point in time later than the point in time given for forecasting. If historical data is available after the given time stamps it is possible to predict failures.
9. Compare the predicted defect or failure rates with the actual number of such incidents and compute the forecast’s relative error.

This process can be repeated for all releases and analyzed to determine the “best” model.

The overall goal is, of course, not to accurately predict the failure rate but to be as close as possible to a distinct margin that is allowed by customer contracts or available maintenance capacity.

VI. ECONOMIC CONSIDERATIONS

Quality improvement is driven by two overarching business objectives, both contributing to increased returns on engineering investments.

- **Improve the customer-perceived quality**. This impacts the top line because sales are increased with satisfied customers.
- **Reduce the total cost of non-quality**. This improves the bottom line, because reduced cost of non-quality means less engineering cost with same output, therefore improved productivity.

Improving customer-perceived quality can be broken down to one major improvement target, to systematically reduce customer detected defects. Reducing field defects and improving customer-perceived quality almost naturally improves the cost-benefit values along the product life-cycle. Investments are made to improve quality which later improves customer satisfaction. Therefore the second objective of reducing the cost of non-quality comes into the picture: this is the cost of activities related to detecting defects too late.

Return on investment (ROI) is a critical, but often misleading expression when it comes to development cost. Too often heterogeneous cost elements with different meaning and unclear accounting relationships are combined into one figure that is then optimized. For instance, reducing the “cost of quality” that includes appraisal cost and prevention cost is misleading when compared with cost of nonconformance because certain appraisal costs (e.g., unit test) are components of regular development. Cost of nonconformance (cost of non-quality) on the other hand is incomplete if we only consider internal cost for defect detection, correction and redelivery because we must include opportunity cost due to rework at the customer site, late deliveries or simply binding resources that otherwise might have been used for a new project.

Not all ROI calculations need to be based immediately on monetary benefits from accounting. Depending on the business goals, they can as well be directly presented in terms of improved delivery accuracy, reduced lead time or higher efficiency and productivity. The latter have a meaning for the market or customer, and thus clearly serve as a ROI basis.

One of the typical improvement objectives (and thus measurements) related to process improvement is reduced cost of non-quality. Reducing cost is for many companies a key business concern and they look for ways how to effectively address sources of cost. Philip Crosby’s cost of quality model provides a useful tool for measuring the impacts of process improvement in this domain [10]. We have extended his model to cover the quality-related cost in software and IT projects [1]. Our model segments cost of building a product and is applicable to all kinds of software and IT products, services and so on. Fig. 6 shows the quality related cost and its elements following four categories.

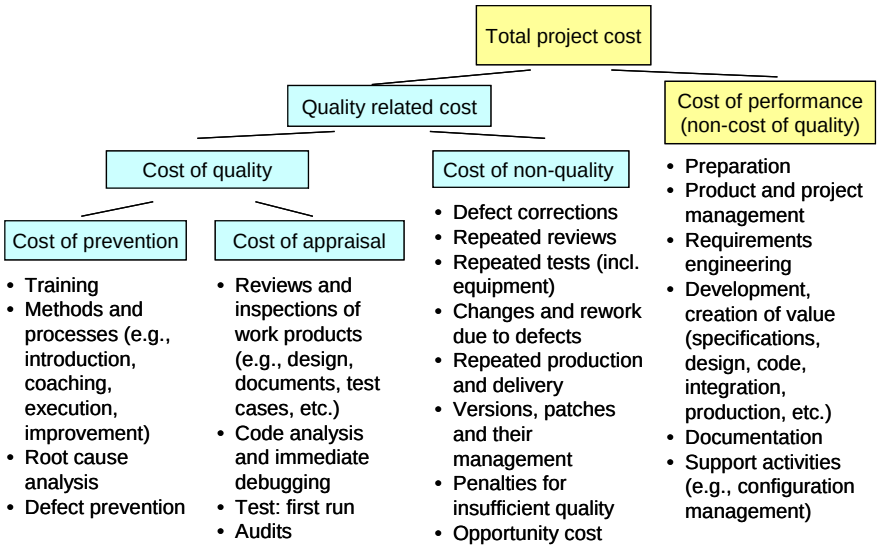

Fig. 6: Quality related cost and its elements

- **Cost of performance or non-cost of quality**. These are any regular cost related to developing and producing the software, such as requirements management, design, documentation or deployment environments. It includes project management, configuration management, tools and so on. It excludes only the following segments which have been defined to distinguish quality-related activities.
- **Cost of prevention**. These are the cost to establish, perform and maintain processes to avoid or reduce the number of defects during the project. It includes dedicated techniques and methods for defect prevention and the related measurement, change management, training and tools.
- **Cost of appraisal**. These are the cost related to detecting and removing defects close to the activity where they had been introduced. This cost typically includes activities such as reviews, unit test, inspections, but also training in these techniques.
- **Cost of nonconformance or cost of non-quality**. These are the cost attributable to not having prevented or removed defects in due time. They can be distinguished towards cost of internal defects (found in testing) and cost of external defects (found after release of the product). Specifically the latter have huge overheads due to providing corrections or even paying penalties. Regression testing, building and deploying patches and corrections, staffing the help desk (for the share of corrective maintenance and service) and re-developing a product or release that misses customer needs are included to cost of non-quality. For conservative reasons (i.e., to not overestimate ROI and savings) they do not include opportunistic cost, such as bad reputation or loosing a contract.

Let us look to a concrete example with empirical evidence. For early defect detection, we will try to provide detailed insight in an ROI calculation based on empirical data from many different industry projects which was collected during the late nineties in numerous studies from a variety of projects of different size and scope [1,24]. Fig. 7 provides data that results from average values that have been

gathered in Alcatel's global project history database over nearly ten years.

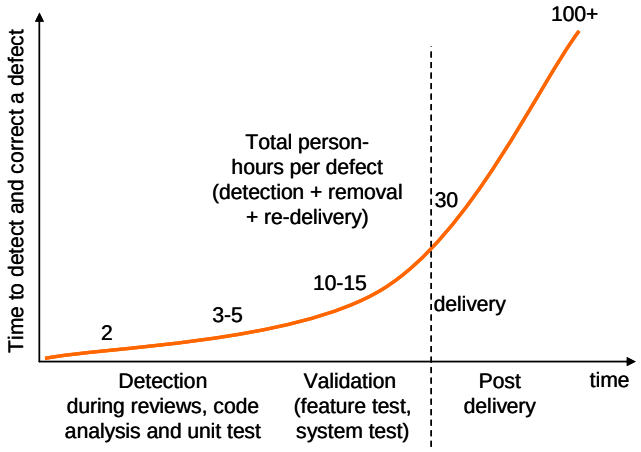

Fig. 7: The defect-related effort to detect, remove and redeliver software depending on the activity where the defects are found

To show the practical impact of early defect removal we will briefly set up the business case. We will compare the effect of increased effort for combined code reading and code inspection activities as a key result of our improvement program. Table 2 shows the summary of raw data that contributes to this business case. The values have been extracted from the global history database for three consecutive years, during which the formalized and systematic code reviews and inspections had been introduced. The baseline column provides the data before starting with systematic code reviews. Year one is the transition time and year two shows the stabilized results. Given an average-sized development project with 75 KStmt and only focusing on the new and changed software without considering any effects of defect-preventive activities over time, the following calculation can be derived. The effort spent for code reading and inspection activities increases by 1470 PH. Assuming a constant average combined appraisal cost and cost of nonperformance (i.e., detection and correction effort) after coding of 15 PH/defect (the value at the time of this analysis in the respective product-line), the total effect is 9030 PH less spent in year 2. This results in a ROI value of 6.1 (i.e., each additional hour spent on code reading and inspections yields 6.1 saved hours of appraisal and nonperformance activities afterwards).

Additional costs for providing the static code analysis and the related statistical analysis are in the range of few person hours per project. The tools used for this exercise are off-the-shelf and readily available (e.g., static code analysis, spreadsheet programs).

The business impact of early defect removal and thus less rework is dramatic. Taking above observations, we can generalize towards a simple rule of thumb. Moving 10% of defects from test activities (with 10-15 PH/defect) to reviews and phase-contained defect removal activities (with 2-3 PH/defect) brings a yield of 3% of total engineering cost reduction.

The calculation is as follows. We assume that an engineer is working for 90% on projects. Of this pro-

ject work, 40% is for testing (which includes all effort related to test activities) and another 40% is for constructive activities (including requirements analysis, specification, design, coding, together with direct phase-contained defect removal). The rest is assumed as overheads related to project management, configuration management, training and so on. So the totality of engineering effort on design and test is 32%, respectively. Obviously the design and test could be done by one or different persons or teams without changing the calculation.

Table 2: The business case for systematic code reviews and inspections

Empirical data, mapped to sample project	baseline	year 1	year 2
Reading speed [Stmt/PH]	183	57	44
Effort in PH per KStmt	15	24	36
Effort in PH per Defect in code reviews	7.5	3	3
Defects per KStmt	2	8	12
Effectiveness [% of all]	2	18	29
Sample project: 70 KStmts. 2100 defects estimated based on 30 defects per KStmt			
Effort for code reading or inspections [PH]	1050		2520
Defects found in code reading/inspections	140		840
Residual defects after code reading/inspections	1960		1260
Correction effort after code reading/inspections [PH] (based on 15 PH/F average correction effort)	29400		18900
Total correction effort [PH]	30450		21420
ROI = saved total effort / additional detection effort			6.1

We assume the design engineer delivers a net amount of 15 KStmt verified code per person year. This amount of code consists only of manually written code, independently whether it is done by means of a programming language or of design languages that would later be translated automatically into code. It does not include reused code or automatically generated code.

Let us assume the person year as 1500 PH. This code contains 20 defects per KStmt of which 50% are found by the designer and 50% by test. So the designer delivers 150 defects per year to test, which cost 10 person hours per defect to remove and re-deliver. This results in an effort of 1.500 PH for testing, which is roughly one person year. Detecting 10% of these defects already during design would number to 150 PH of saved test effort assuming that test cases would be decreased, which is normally the case once the input stability is improving. These 10% of defects would cost 2 person hours each for additional verification effort, totaling to 30 PH. The savings would be 120 PH which we can compare to our engineering cost for the 15 delivered KStmt.

Original cost of the 15 delivered KStmt of code is one person year of design and one person year of test. This accumulates to 3000 PH. With the change, the total cost would be 1530 PH (design plus additional verification) plus 1350 PH (new test effort), which equals 2880 person hours. The 10% move of defects from test to verification total a saving of 120 PH which is 4% of the respective workload be-

ing saved. We have reduced net engineering cost by 4% by detecting an additional 10% of defects before test!

Taking into consideration the gross cost of engineering, which allows only 90% of engineering time spent on project work and only 80% of that time spent for design and test within the project-related effort means that 72% of engineering time is directly attributable to code and test. 72% multiplied with the 4% savings above results in a gross saving of 2.9%.

This means that from a total engineering expense perspective, moving 10% of defects from test to design yields a benefit of 3% to overall engineering cost. Early defect removal is a key element of any efficiency improvement – much more reliable than offshoring.

Note that we took a conservative approach to this savings calculation, by for instance leaving out any cost of defects found after delivery to the customer or opportunistic effects from a not so satisfied customer or from additional service cost. You should certainly make a similar calculation in your own environment to judge the business impact of your change and improvement initiatives.

Motorola has done similar studies during the past years and correlated the quality related cost (for definitions see Fig. 6) with the achieved CMMI maturity level [25]. Fig. 8 shows the results starting with maturity level 1 and highest total cost of quality on the left side. The impacts and culture changes obtained with higher maturity are provided by moving to the right side. It ends with maturity level 5 and lowest overall cost of software quality. Again we see what we stressed already in the introduction, namely that preventive activities yield benefits only if quality control and quality assurance activities are well under control. Quality management has to be engrained in the company's culture which takes some time. Moving from one to the next maturity level typically takes one to two years depending on the size of the company and its organizational complexities.

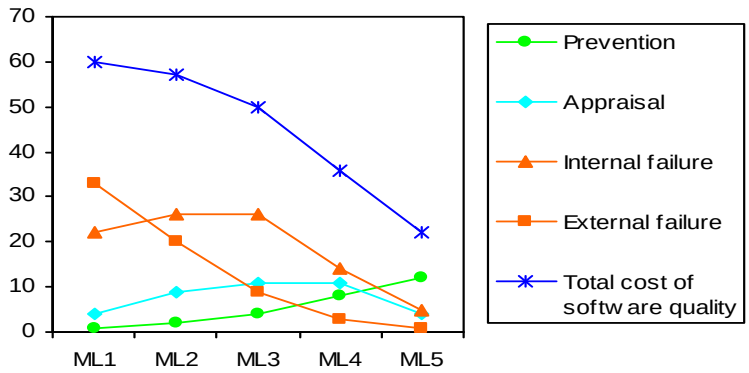

Fig. 8: Improving process maturity at Motorola reduced the total cost of quality

There are many similar observations in software engineering that the author has observed throughout many years of working with very different engineering teams, working on all types of software [1,20,24,26,27,28]. Generally it pays off to remove defects close to the phase where they are inserted. Or more generally to remove accidental activities and outcome, while controlling essential activities.

We assert this observation and the impacts of early defect removal to a law which states in simple words: **Productivity is improved by reducing accidents and controlling essence** [1]. This law can be abbreviated with the first letters of the four words to a simple, yet powerful word: RACE (Reduce Accidents, Control Essence). Those who succeed with RACE will have not only cost savings but also be in time and thus win the race on market share. Early defect removal, that is reduced accidents, and focus on what matters, that is control essence means productivity and time to profit!

VII. SOFTWARE QUALITY RULES OF THUMB

This chapter will summarize the quantifiable experiences and wisdom that the author collected in his over 20 years of practical software engineering. The list is far from complete and certainly is not as scientific as one would like – but it is a start. The data stems from our own history databases [1]. Clearly, this is not a substitute for your own quality database. You need to build over time your own experience database with baselines for estimation, quality planning and the like. However, you might not yet have this data available, or it is not yet scaleable for new products, methodologies or projects. Knowing that it is often difficult to just use plain numbers to characterize a situation, we are also aware that beginners and practitioners need some numbers to build upon – even if their applicability is somewhat limited. We will therefore provide in this chapter concrete and fact-based guidance with numbers from our own experiences so you can use it as a baselines in your projects.

The **number of defects at code completion** (i.e., after coding has been finished for a specific component and has passed compilation) can be estimated in different ways. If size in KStmt or KLOC is known, this can be translated into residual defects. We found some 10-50 defects per KStmt depending on the maturity level of the respective organization [1]. IFPUG uses the predictor of 1.2 defects per function point which translates for C-language into 20 defects per KStmt [29]. Alternatively it is recommended for bigger projects to calculate defects as $FP^{1.2}$. This is based only on new or changed code, not including any code that is reused or automatically generated. For such code, the initial formula has to be extended with percentage of defects found by the already completed verification (or validation) steps. An alternative formula for new projects takes estimated function points of a project to the power of 1.25 [1].

Verification pays off. Peer reviews and inspections are the least expensive of all manual defect detection techniques. You need some 1-3 person hours per defect for inspections and peer reviews [1]. Before starting peer reviews or inspections, all tool-supported techniques should be fully exploited, such as static and dynamic checking of source code. Fully instrumented unit test should preferably be done before peer reviews. Unit test, static code analysis and peer reviews are orthogonal techniques that detect different defect classes. Often cost per defect in unit test is the highest amongst the three techniques due to the manual handling of test stubs, test environments, test cases, and so on.

Defect phase containment has clear business impact. Detecting 10% more defects in design or code

reviews and therefore reducing test effort and long rework cycles yields a savings potential of 3% of engineering cost [1].

Cost of non-quality (i.e., defect detection and correction after the activity where the defect was introduced) is around 30-50% of total engineering (project) effort [1]. A significant percentage of the effort on current software projects is typically spent on avoidable rework [20]. It is by far the biggest chunk in any project that can be reduced to directly and immediately save cost! Especially for global software engineering projects, this cost rather increases due to interface overheads where code or design would be shipped back and forth until defects are retrieved and removed. The amount of effort spent on avoidable rework decreases as process maturity increases [1,20].

Typically **testing** consumes more than 40% of the resources and – depending on the project life-cycle (sequential or incremental) – a lead-time of 15-50% compared to total project duration [1]. The minimum 15% lead-time is achieved when test strongly overlaps with development, such as in incremental development with a stable build which is continuously regression tested. In such case, there is only the system test at the end contributing to lead-time on the critical project path. On the other hand 50% (and more) stem from testing practiced in a classic waterfall approach with lots of overheads due to components that won't integrate.

Cost of defects after delivery. Finding and fixing a severe software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase [1,20]. This relates to the cost of rework and correction which increases fast once the software system is built and delivered.

Each **verification or validation step** as a rule of thumb will detect and remove around 30% of the then residual defects [1]. This means that 30% of defects remaining at a certain point of time can be found with a distinct defect detection technique. This is a cascading approach, where each cascade (e.g., static checking, peer review, unit test, integration test, system test, beta test) removes each 30% of defects. It is possible to exceed this number slightly towards 40-50% but at steeply increasing cost per defect. Reviews of work products can catch more than half of a product's defects regardless of the domain, level of maturity of the organization, or life-cycle phase during which they were applied [20]. It is however important to realize that each such activity has an efficiency optimum (see Table 3) [1]. Going beyond that optimum often means an increasing cost per defect removed. It could still be valid doing so, especially for critical software, but a careful planning is required to optimize total cost of non-quality.

Table 3: Defect detection rate per activity compared to the total number of defects

Project activities	Maximum	Typical defect detection effectiveness	Insufficient
Requirements reviews	10-15%	5-10%	0%
Design reviews	10%	5-10%	0%
Code: static code analysis	20%	10-20%	<10%
Code: peer reviews and inspections	40%	20-30%	<10%
Code: code analysis and unit test	30%	10-30%	<10%
Integration test	20%	5-20%	>20%
Qualification / release test	5%	1-5%	>5%
Total percentage removed	100%	95-98%	<90%

Residual defects are estimated from estimated total defects and the different detected defects. This allows the planning of verification and validation and of allocating necessary time and budget according to quality needs. If 30% of the defects are removed per detection activity then 70% will remain. Residual defects at the end of the project thus equal the number of defects at code completion times 70% to the power of independent detection activities (e.g., code inspection, module test, integration test, system test, and so on).

Release quality of software shows that typically 10% of the initial defects at code completion will reach the customer [1]. Depending on the maturity of the software organization, the following number of defects at release time can be observed [1]:

- CMMI maturity level 1: 5-60 defects/KStmt
- Maturity level 2: 3-12 defects/KStmt
- Maturity level 3: 2-7 defects/KStmt
- Maturity level 4: 1-5 defects/KStmt
- Maturity level 5: 0.05-1 defects/KStmt.

Quality of external components from suppliers on low process maturity levels is typically poor. Suppliers with high maturity (i.e., on or above CMMI maturity level 3) will have acceptable defect rates, but only if they own the entire product or component and manage their own suppliers. Virtual (globally distributed) development demands more quality control and thus cost of quality to achieve the same release quality.

Improving release quality needs time: 5% more defects detected before release time translates into 10-15% more duration of the project [1].

New defects are inserted with changes and corrections, specifically those late in a project and done under pressure. Corrections create some 5-30% new defects depending on time pressure and underlying tool support [1]. Especially late defect removal while being on the project's critical path to release causes many new defects with any change or correction because quality assurance activities are reduced and engineers are stressed. This must be considered when planning testing, validation or

maintenance activities.

The Pareto principle also holds for software engineering [1]:

- 10% of all code account for 90% of outage time
- As a rule of thumb, 20% of all components (subsystems, modules, classes) consume 60-80% of all resources.
- 60% of a system's defects come from 20% of its components (modules, classes, units). However, the distribution varies based on environment characteristics such as processes used and quality goals.
- Post-release, about 40% of modules may be defect-free [20].
- 20% of all defects need 60-80% of correction effort. Most of the avoidable rework comes from a small number of software defects, where avoidable rework is defined as work done to mitigate the effects of errors or to improve system performance [20].
- 20% of all enhancements require 60-80% of all maintenance effort.

This might appear a bit theoretical because obviously Pareto distributions rule our world – not only that of software engineering. It is always the few relevant members in any set which govern the set's behaviors. However, there are concrete, practically useful benefits you can utilize to save on effort. For instance, critical components can be identified in the design by static code analysis and verification activities then can be focused on those critical components.

VIII. CONCLUSIONS

The article has introduced to software quality management. It detailed basic ingredients of quality management, such as quality policies, quality control and quality assurance. It introduced to major standards and detailed why good processes and engrained discipline in engineering and management processes will drive better quality. Focus was given to the concept of defects and how they can be predicted, detected, removed and prevented. Quality is free, but you need to invest in some place in order to earn more money in the bottom line. This relationship of investment in better people, processes and discipline was outlined by providing some sample business cases related to introducing quality management.

Let us conclude with a set of practical advice that help in building a quality culture:

- Specify early in the product life-cycle and certainly before start of a project what quality level needs to be achieved. State concrete and measurable quality requirements. Ensure that the quality objectives are committed for the product and project. Agree within the project plan the respective techniques and processes to achieve these committed quality objectives.
- Establish the notion of "good enough". Avoid demanding too much quality. Quality objectives have to balance market and internal needs with the cost and effort to achieve them. It is of no benefit to shoot for perfectionism if the market is not willing to pay for it. It is however equally

stupid to deliver insufficient quality and thus loose reputation. Note that bad quality will spread and multiply in its effect much faster and wider than delivering according to specification or above.

- Clearly distinguish quality control activities from quality assurance activities. They are both necessary but done by different parties. Quality control is the responsibility of each single employee delivering or handling (work) products. Quality assurance comes on top and is based on samples to check whether processes and work products conform to requirements.
- Implement the combination of “built-in” and “bolt-on” attributes of quality. The approach of combination of prevention and early detection of defects will cut the risk of defects found by the customer.
- Leverage management and staff with previous quality assurance experience to accelerate change management.
- Measure and monitor quality consistently from requirement definition to deployment. This will help in keeping the efforts focused across the full life-cycle of the product development. Use measurements for effectiveness (i.e., how many defects are found or what percentage of defects are found or what type of defects are found with a distinct defect removal activity) and efficiency (i.e., how much effort per defect removal or how much effort per test case, cost of non-quality) to show the value of early defect removal. Convince your senior management of the business case and they will be your best friends in pushing enough resources into development to contain defect removal to the phase where defects are introduced. Only old-fashioned managers like test (because they think it is the way to see the software working), modern managers just hate test due to its high cost, insufficient controllability and poor effectiveness.
- Know about defects, where they origin, how to best detect them and how to remove them most effectively and efficiently. Do not be satisfied with removing defects. Testing is a big waste of effort and should be reduced as much as possible. With each defect found think about how to change the process to avoid the defect from reoccurring. Defects should be removed in the phase where they are created. Being successful with defect phase containment needs few practical prerequisites.
- Ensure the right level of discipline. This includes planning, preparation, training, checklists, monitoring. Too much formalism can cause inefficiencies, not enough formalism will reduce effectiveness. Reporting of defects and effort per defect is key to optimize the process and forecast residual defects. Without reporting, reviews are not worth the effort.
- Apply different types of early defect correction such as reviews, inspections, tool-based software analysis and unit test. Requirements inspections are mandatory for all projects and should be done by designers, testers and product managers. Hardware is analyzed by tools and inspected with specific guidelines. GUI design (e.g., Web pages) are reviewed for usability and so on. Software design and code are statically analyzed plus inspected. Inspections should focus on

critical areas with more defects than average. Critical areas can be identified based on complexity, change history and so on.

- Measure and improve engineering efficiency by continuously measuring software quality and its different cost parameters. Verification techniques should find 50-60% of the residual defects. They are typically cascaded to detect different defect types. Detecting 10% more defects in design or code reviews and therefore reducing test effort and long rework cycles yields a savings potential of 3% of engineering cost.
- Follow a structured and systematic methodology as demanded by CMMI, Six Sigma, ISO 9001 etc. as the basis for process improvement. Once the processes are in place, measure their performance, reduce variation and define strategy for continuous improvement.

Quality is about people and processes. Never forget about the people factor. It is engineers and management that drive culture in a software company. Include training and development on quality control and quality assurance procedures for every employee as a part of their role and incentives. Assure that especially management would walk the talk. Or in the words of M. Gandhi: “You must be the change that you want to see in the world”.

Acknowledgements

Some parts of sections III, IV, V, VI and VII appeared first in Ebert, C., and R.Dumke: Software Measurement. Copyrights: Springer, Heidelberg, New York, 2007. Used with permission. We recommend reading that book as an extension of the quantitative concepts mentioned in this article.

Further Reading

There are few basic references that we recommend to get a good understanding of quality management. A very basic reading is the classic text of Crosby [10]. From a formal perspective, obviously the ISO 9001 standard [3] is an absolute prerequisite to understand quality management. Software quality improvement is often build upon the CMMI as a framework for process improvement. We recommend reading the underlying reference of Chrissis et al [5]. Those who want to embark further in reliability engineering should read the books of Musa and Lyu [7,8]. Quantitative techniques and measurement concepts and experiences are detailed in the authors book on software measurement [1]. A sound reference for lots of quantitative data is the book of Jones [16]. A look into how quality concepts relate to liability and legal aspects is provided in a Vector Consulting Services white paper [30].

Literature

[1] Ebert, C. and R. Dumke: Software Measurement. Springer, Heidelberg, New York, 2007.
[2] Buzzel, R. D. and B. T. Gale: The PIMS Principles – Linking Strategy to Performance. The Free Press, New York, 1987.
[3] ISO 9001:2000. Quality management systems – Requirements. ISO, http://www.iso.org (2000)
[4] CMMI® for Development, Version 1.2. CMU/SEI-2006-TR-008. August 2006

[5] Chrissis, M.B., M.Konrad and S.Shrum: CMMI. Guidelines for Process Integration and Product Improvement, ed. 2. Addison-Wesley, Reading, USA (2006)
[6] Khoshgoftaar, T.M. et al: Early Quality Prediction: A Case Study in Telecommunications. IEEE Software, 13: 65-71 (1996)
[7] Musa, J. D., Iannino, A., Okumoto, K.: Software Reliability – Measurement, Prediction, Application. McGraw-Hill, New York, (1987)
[8] Lyu, M.R.: Handbook of Software Reliability Engineering. McGraw-Hill, New York, (1995)
[9] McConnell, S.: Professional Software Development. Addison-Wesley, Reading, USA, (2003)
[10] Crosby, Philip: Quality is Free. New American Library, New York, 1979.
[11] Humphrey, W. S.: Managing the Software Process. Addison-Wesley, Reading, USA (1989)
[12] ISO/IEC 15504-2:2003. Information technology – Process assessment – Part 2: Performing an assessment. ISO, http://www.iso.org (2003)
[32] Performance Results of CMMI®-Based Process Improvement. CMU/SEI-2006-TR-004. August 2006.
[14] Cai, K.: On Estimating the Number of Defects Remaining in Software. The Journal of Systems and Software. Vol. 40, No. 2, pp. 93-114 (1998)
[15] Mills, H. D.: On the Statistical Validation of Computer Programs. Technical Report FSC-72-6015, Gaithersburg, MD : IBM Federal Systems Division (1972)
[16] Jones, C., Software Quality, International Thomson Computer Press, Boston, MA, (1997)
[17] Humphrey, W. S.: Introduction to the Personal Software Process. Addison-Wesley, Reading, USA (1997)
[18] Jones, C., Applied Software Measurement, McGraw-Hill, New York, (1996)
[19] Wayne, M. Z., Zage, D. M.: Evaluating Design Metrics on Large-Scale Software. IEEE Software, Vol. 10, No. 7, pp. 75–81, Jul. 1993 (1993)
[20] Shull, F. et al: What we have learned about fighting defects. Proceedings of the 8th International Symposium on Software Metrics. IEEE, Los Alamitos, USA, pp. 249-258 (2002)
[21] Ebert, C.: Tracing Complexity through the Software Process. Proc. Int. Conf. on Engineering of Complex Computer Systems ICECCS'95. IEEE Computer Soc. Press., pp. 23-30, Los Alamitos, CA, USA, 1995.
[22] Evanco, W. M. and W. W, Agresti: A composite complexity approach for software defect modeling. Software Quality Journal, 3: pp. 27-44 (1994)
[23] Musa, J. D., Iannino, A.: Estimating the Total Number of Software Failures Using an Exponential Model. Software Engineering Notes, Vol. 16, No. 3, pp. 1-10, July 1991 (1991)
[24] Ebert, C., T.Liedtke, E.Baisch: Improving Reliability of Large Software Systems. In: A.L.Goel, ed.: Annals of Software Engineering. 8: pp. 3-51 (1999)
[25] Eickelmann, N.: Measuring Maturity Goes beyond Process. IEEE Software, Vol. 21, No. 4, pg. 12-13 (2004)
[26] Ebert, C.: Improving Validation Activities in a Global Software Development. Proc. Int. Conf. on Software Engineering 2001. IEEE Comp. Soc. Press, Los Alamitos, USA, 2001
[27] Ebert, C.: Global Software Engineering. IEEE ReadyNote (e-Book), IEEE Computer Society, Los Alamitos, USA, 2006.
http://www.computer.org/portal/cms_docs_cs/ieeecs/jsp/ReadyNotes/displayRNCatalog.jsp . Cited 25. July 2007
[28] Ebert, C.: Understanding the Product Life-cycle: Four Key Requirements Engineering Techniques. IEEE Software, Vol. 23, No. 3, pp. 19-25, May 2006
[29] Longstreet, D.: Test Cases & Defects. Published at http://www.ifpug.com/Articles/defects.htm. Cited 25. July 2007.
[30] Amsler, K. et al: Safety-critical engineering processes. Vector Consulting, White Paper, 2004.
http://www.vector-consulting.de/portal/datei_mediendatenbank.php?system_id=106900 Cited 25. July 2007.