Chapter 1

Why Software Quality Engineering?

Quality has become a critical attribute of software products as its absence produces financial, health, and sometimes life losses. At the same time the definition, or scope, of the domain of software quality has evolved continuously from a somewhat technical perspective to a perspective that embraces human aspects such as usability and satisfaction.

An increasing business-related recognition of the importance of software quality has also made software engineering's "center of gravity" shift from *creating an engineering solution* toward *satisfying the stakeholders*. Such a shift very clearly reflects the trend within the community of stakeholders who more and more often say: "I do not want to know about *bits and bytes*. I want a solution that satisfies my needs." The critical word here is "satisfaction," for it covers both functional and quality perception of the software solution being used.

Development organizations confronted with such an approach are, in general, not entirely prepared to deal with it even if their engineers are adequately educated. Moreover, if the education is there, it is quite often acquired through experience rather than a regular educational process, as the software engineering curricula being offered, with few exceptions [1], do not emphasize the importance of teaching software quality engineering.

One of practical responses to such a situation was the development of Software Engineering Body of Knowledge (SWEBOK) [2]. SWEBOK seeks to provide the knowledge that allows universities to build such educational programs that will allow producing professionals able to stay abreast of the fast-moving industry, but it also adds a scientific and innovative component to *best practices*. The continuation of this approach is this book.

So let's ask the question, "why software quality engineering?" as three partial questions:

Software Quality Engineering: A Practitioner's Approach, First Edition. Witold Suryn.

Published 2014 by John Wiley & Sons, Inc.

- *Why software?* Because in contemporary social life software, systems and services rendered by software are omnipresent, beginning with the watches we wear, ending with nuclear electricity plants or spaceships.
- *Why quality?* Because if these instances of software work without the required quality we may be late, dead, or lost in space.
- *Why engineering?* As in every technical domain, it is engineering that transforms ideas into products, it is the verified and validated set of "to-dos" that help develop the product that not only has required functionalities but also executes them correctly.

To make this picture complete, another question should be asked: *why at all?* There is in fact only one reason: the user. Despite decades of evolution of information technology and its tools, the user still faces risky, unreliable, and quite often unintelligent products that far too often waste his or her time or money, and wear off his or her patience. So quality engineering applied to software, systems, and related services is intended to assist developers in building good, intelligent, and reliable products; to help users request and verify their quality needs; and for those who want to use software as easily as they use a dishwasher, to shield against faulty products and unprofessional suppliers.

1.1 SOFTWARE QUALITY IN THE REAL WORLD

For the users, a software product more and more often corresponds to a black box that must effectively support their business processes. As a consequence of this natural approach, business needs become a driving force of quality software product development and a stakeholder moves to the position of a car buyer and user rather than an involuntary expert in software engineering. And what he or she perceives at the end corresponds to expressed satisfaction at using a software product that possesses *both* required functionalities and required quality. When one of them is missing, a painful process of improvements and negotiations takes place to often end by changing the supplier and replacing the product with one that is *mature* enough do its job well on both accounts.

What exactly constitutes the quality of a product is often the subject of hot debate. The reason the concept of quality is so controversial is that there is no common agreement on what it means. For some it is "degree to which a set of inherent characteristics fulfills requirements" [3], whereas for others it can be synonymous with "customer value," or even "defect levels" [4]. A possible explanation as to why any of these definitions could not win a consensus is that they generally do not recognize different perspectives of quality, such as for instance the five proposed by Kitchenham and Pfleeger [5]:

- The transcendental perspective deals with the metaphysical aspect of quality. In this view of quality, it is "something toward which we strive as an ideal, but may never implement completely."

- The user perspective is concerned with the appropriateness of the product for a given context of use.
- The manufacturing perspective represents quality as conformance to requirements. This aspect of quality is stressed by standards such as ISO 9001 [6] or models such as the Capability Maturity Model [7].
- The product perspective implies that quality can be appreciated by measuring the inherent characteristics of the product.
- The final perspective of quality is value-based. This perspective recognizes that the different perspectives of quality may have a different importance, or value, to various stakeholders.

One could argue that in a world where conformance to ISO and IEEE standards is increasingly present in contractual agreements and used as a marketing tool, all the perspectives of quality are subordinate to the manufacturing view. This predominance of the manufacturing view in software engineering can be traced back to the 1960s, when the U.S. Department of Defense and IBM gave birth to Software Quality Assurance [8]. This has led to the belief that adherence to a development process, as in manufacturing, will lead to a quality product. The corollary to this belief is that process improvement will lead to improved product quality.

This opinion is not shared unanimously, as some parts of both industry and academia find it inaccurate or at least flawed. For example, G. Dromey states:

> The flaw in this approach [that you need a quality process to produce a quality product] is that the emphasis on process usually comes at the expense of constructing, refining, and using adequate product quality models [9].

Kitchenham and Pfleeger reinforce this opinion by stating:

> There is little evidence that conformance to process standards guarantees good products. In fact, the critics of this view suggest that process standards guarantee only uniformity of output [5].

Furthermore, data available from Agile [4] projects show that high quality is attainable without following a manufacturing-like approach.

However, some studies conducted at Raytheon [10] and Motorola [11] showed that there is indeed a correlation between the maturity level of an organization as measured by the Capability Maturity Model (CMM) and the quality of the resulting product. These studies provide data on how a higher maturity level (as measured by the CMM) can lead to:

- Improved error/defect density (i.e., the error/defect density lowers as maturity improves)
- Lower error rate
- Lower cycle time (time to complete parts of the lifecycle)
- Better estimation capability.

From these results, one could conclude the quality can be improved by following a mature process. Studies of the development of lifecycle models presented by Georgiadou [12] indicate that the maturity of the development process is reflected by the emphasis and allocation of testing and other quality assurance activities. The study demonstrated that the more mature the process and its underlying life cycle model, the earlier the identification of errors in the deliverables. However, these measured improvements are directly related to the manufacturing perspective of quality. Therefore, such quality improvement efforts fail to address the other perspectives of quality. This might be one of the reasons for the perception of the "quality problem" as one of the main failings of the software engineering industry. Furthermore, studies show that improvement efforts rooted in the manufacturing perspective of quality are difficult to scale down to smaller projects and/or smaller teams [13, 14]. Indeed, rather than being scaled down in smaller projects, these practices tend to be not performed at all.

Over recent years, researchers have proposed new approaches and models that try to encompass more perspectives of quality than just the manufacturing view. Geoff Dromey [9, 15] proposed such a model in which the quality of the end product is directly related to the quality of the artifacts that are a by-product of the process being followed. The reasoning is that if quality artifacts are correctly designed and produced throughout the life cycle, then the end product shall manifest attributes of good quality. This approach can clearly be linked to the product perspective of quality with elements from the manufacturing view. This is certainly a step from the manufacturing-only approach, but it fails to view the engineering of quality as a process that covers all the perspectives of quality. In Pfleeger and Atlee [16], the reader can find valid arguments against approaches that focus only on the product perspective of quality:

> This view [the product view] is the one often advocated by software metrics experts; they assume that good internal quality indicators will lead to good external ones, such as reliability and maintainability. However, more research is needed to verify these assumptions and to determine which aspects of quality affect the actual product's use.

All of this may be true to a certain extent, but what ultimately counts is a customer's *yes* said after the delivery is finalized.

Another absolutely natural trend observable within the "population of IT customers" is the desire to be properly served without having to become proficient in information technology. A customer just wants to buy, learn how to use, and then simply use a software product, just as he or she does with a car or a TV. This boils down to an extended (or shall we just say "professional and mature") responsibility of a software supplier, who now has to know not only what the customer is able to express, but also what the customer does not know that he or she knows. And then, when all questions are asked and answered, the supplier must continue on his or her way until the product is built and delivered to the customer's satisfaction.

Similarly to mathematics, the most important part of software and software quality engineering is to understand the problem. Whatever comes after is the result

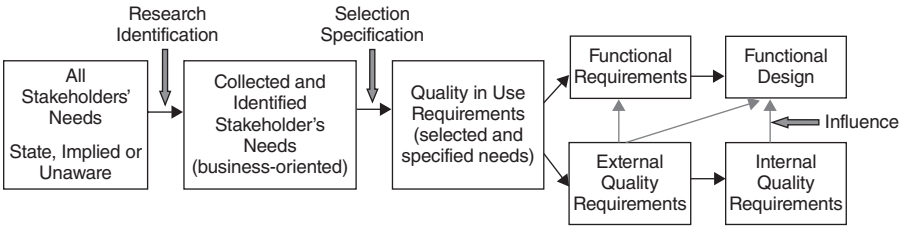

Figure 1.1 From "stated, implied, and unaware" needs to fully defined software product (based on personal communication of M. Azuma).

of knowledge applied to this understanding and, if we make an assumption that such knowledge exists, the final outcome makes the "executable form" of what was understood. The graveness of this statement is expressed by different kinds of statistics showing billion-dollar losses resulting from bad or incomplete understanding of the problem called a software product (a simple search on Google brings up thousands of hits on this subject). In case of software quality, the situation is even more dire, as the primary source of information, a customer, is usually able to at least signal his or her "functional" needs, but in the majority of situations is not knowledgeable enough to identify or discuss in precise terms the quality requested from the product under discussion. When it comes then to analyzing why something bad happened, customers blame suppliers (which is understandable) but the suppliers do not stay behind. From a purely professional point of view, one might ask: "Is that fair?" Who, between the user and the supplier, is supposed to be an expert, especially in a subject so difficult to define as quality? Should not it be the supplier who follows the process from Fig. 1.1 (with the customer having his or her "stated, implied or unaware" needs), in order to solicit, identify, and define required quality attributes and then later develop a software product that exhibits them? This question is a keynote and the main subject of this book.

1.1.1 Consumer Perspective

When a car manufacturer asks a customer about his or her opinion on the vehicle the latter uses or was using, the manufacturer, in fact, asks about an overall perception of the car in question, including both functionalities and the quality associated with them. In the case of software and even more in the case of software systems, the overall perception (or satisfaction) is heavily influenced by the verifiable existence of quality. During his many years of working in the IT industry and then teaching at a university, the author had the unique chance to ask the following question to IT professionals, students, and customers: "What would you be more inclined to accept, a system with a rich set of functionalities but with lower quality or the one with limited functionalities but with high quality?" The choice was in 99% of cases the same: the second. Interesting that the choice was identical even

if the interviewed persons were from different sides of the IT market "barricade," suppliers and users. Obviously, the choice becomes less firm when suppliers return to their workstations (or we would have only high quality and bug-free software) but still, such unanimity may be interpreted as a good sign. The choice may sound "generic" as the reaction, but its real context varies for a supplier and a consumer and even inside these categories, as in the case of an individual and a corporate consumer.

1.1.1.1 Individual Consumer

In the majority of cases the individual consumer is a person with no face and no name. Unfortunately, the consumer quite often has no rights, too. The simple fact that almost every software on the planet before installation requires the acceptance of license terms that virtually free the manufacturer from any responsibility makes the existence of quality an extra effort that has in mind the good reputation of the supplier rather than the well being of the user. Currently, no known legal case initiated by an individual user against an IT giant has been won. The most common individual user reaction to a software malfunction is "reboot, and pray it works a bit longer." So, in a way, it is the user who is responsible for his or her own misery, for instead of (massively) protesting, even suing, the consumer tends to sit tight and stay quiet. There is also another perspective from which the subject may be looked at: how big is the population of faceless and disenfranchised users who experience serious troubles with individual user-targeted software? How many of us stretch the application to its limits and how many just float in the main and central current of available functionalities? From what we may observe, the latter category is dominant, or the risk of huge financial losses would motivate the suppliers better. What could (or should) be done then to assure the minimal, acceptable quality of *any* software for a Mr. John Doe? One of the emerging options is the *certification* of IT products for the individual user market. The real value of the certification is however linked to the existence of real consequences, be they financial, legal, or even only hitting someone's reputation. If the customer was inclined *not* to buy a noncertified IT product, the supplier would be motivated enough to see quality as an obligation, not as an option. The certification itself could be applied *de jure* or *de facto*, depending on the level of pressure a given society would decide to apply. One of the most important aspects of "quality for John Doe" is the identification and definition of what exactly constitutes the *minimal, acceptable quality level*. This definition would then become a pass/fail criterion used in the certification process. The very interesting beginnings of activities aiming to increase individual consumer IT products' quality can be observed in several countries such as France (Infocert [17]) and Poland (SASO [18]), or on an international level (Quality Assurance Institute [19]). Even if none of them is officially sanctioned as government requested, the market itself reacted in surprisingly positive way. In case of SASO Poland, several local IT corporations requested the possibility to begin certification process, finding it the obvious option for proving the reliability of their products and, in consequence, enhancing their market reach.

1.1.1.2 Corporate Consumer

Corporate consumers may not always have one, identifiable face but in most cases they have recognizable power to demand and obtain. The fact of being "corporate" does not limit this category of IT customers to using only big IT structures, be it a system developed on demand or an individualized suite (like the ones from SAP or Oracle). On the contrary, simpler office applications play a substantial role in corporate world even if they are not used to serve business-critical processes.

From the perspective of IT proficiency, corporate consumers may be put in two distinctive categories: pure users and user-operators. Pure users are those who make the customers of system integration organizations (SIOs) such as HP Enterprise Services (formerly Electronic Data Systems (EDS)) or Oracle. Their business philosophy is "focus on what we know how to do and pay for required specialized services." In many cases the corporate customer of an SIO not only pays for the system, its installation, and required user training, but also pays for further operation and maintenance. Such a business arrangement, popularly known as *outsourcing*, seems to be a win-win solution for all involved. In theory everybody does what he or she knows best; the user focuses on his or her core business without the burden of having his or her own IT team, and the SIO runs the system with all required professionalism and responsibility. What is the place of (in this case) system quality engineering in it?

The simple fact of separating business processes and activities from the running IT machine that supports them puts the whole quality engineering responsibility on the side of a supplier (e.g., an SIO). The customer pays, among other things, to be able to express his or her needs and to be correctly understood using principally the taxonomy natural to his or her business. In consequence, a somehow trivial statement of "I will open a new facility in Japan that has to operate 24/7" will have to be translated into a set of precise functional and quality requirements for the supporting IT system by a supplier, who further should initiate a series of technical meetings where the customer's functional and quality needs are explained, negotiated, understood, and finally agreed upon. The difficulty of this challenge gets bigger when a discussion of quality takes place. Although questions concerning functional aspects of the system are usually easily understood and answered by an IT-unfamiliar user, a question such as "what are your usability requirements?" may raise a few brows. So the supplier not only has to identify his or her customer's quality requirements, but also has to explain them, verify them, and get the fully informed customer's approval, and then engineer them into the system.

The corporate customer's supplier's responsibility does not end with installing the system, training the staff, and turning the key in the ignition. As the parties are known by name and bound by elaborate contracts, the repercussions of missing quality may be traced back and legal and financial consequences can eventually be imposed on the guilty party. If an *outsourcing* contract has been signed, the responsibility for the system and its quality stays on the side of the supplier for the length of the contract.

In case of user-operators the quality engineering problem may be slightly less difficult, as this category of corporate customers is usually "IT-savvy." The biggest

challenge in the whole process of engineering quality, the identification and definition of quality requirements, may eventually be achieved through discussions in domain-specific language and applying domain-specific models and knowledge (e.g., using the ISO/IEC 25000 series of standards [20]), so the road to a correct understanding of quality needs for the given system is shorter and faster.

Then, after the installation and all required training, the system usually goes under the operation and daily maintenance of the user's IT team, with the supplier granting the support and warranty for a given period. Analyzing the responsibility for quality engineering in this type of situation brings a three-phase view: in the phase of the development and transition, it is supplier's sole responsibility; in the phase of the user's operation covered by supplier's warranty, the responsibility is "distributed" and creates the majority of conflicting situations because there is more than one entity manipulating the system; and in the phase of the whole remaining system life time the responsibility for engineering quality is entirely the user's.

One may ask, "what engineering of the quality may take place when system is in its operation phase?" More of this subject will be discussed in Chapter 2.

1.1.2 Supplier Perspective

The supplier's ultimate justification for developing any product is the profit, usually calculated in terms of the return on investment (ROI). It is widely known and accepted that developing functionalities of the system or software requires appropriate budget, but it is much less publicly obvious that engineering the quality into these functionalities costs money as well, and that it is not cheap. There is another aspect of quality that makes it "a child of a lesser god" in eyes of a developer: too often its presence or absence manifests itself after a considerably long time of operation. With the quality of a system or software it is like with a pair of shoes: their "functionalities," such as shape, color, and size, can be seen immediately, but verifying their "qualities," such as real quality of materials used or comfort in use, requires time and operation (walking a few kilometers) to be applied.

These two elements make up the basic reasoning for quality-related decisions. In other words, if the quality is so expensive that it will make the price prohibitive or eat up the profit, it will be reduced to a passable minimum. If, further in this direction, its lack will not be immediately noticed or will not reach the "pain threshold" of the user, it will also be reduced or even neglected. The third element in quality-related decision making is the famous *time-to-market*, the offspring of competition. On one hand, the competition makes a supplier try to build a better product than the other suppliers, but on the other hand, it creates a strong time pressure to reach the market before the competitors, and that always requires compromises. Depending on the corporate philosophy and culture of the supplier, the compromises may be applied to both functionalities and quality or to quality only.

Financial influences are not the only ones that decide the final quality of a software product or system. Quality requires engineering knowledge comparable to that used in development, but this knowledge is far younger and still in dynamic

evolution. In Chapter 2, quality engineering processes and activities are discussed in detail, but to create a simple, common reference for the two following chapters, these processes are named here:

- Identification and definition of quality requirements
- Transformation of requirements into quality attributes of the future software or system
- Transformation of quality attributes into engineering "to-dos" that can be communicated to developers and further realized
- Identification and estimation of interdependencies between development and quality engineering activities
- Design of quality measurement (design of quality tests)
- Quality measurement
- Quality evaluation.

In conclusion, the supplier's perspective on quality engineering is the result of a combination of financial constraints, software quality engineering knowledge existing in the organization, and the user's tolerance to poor quality.

1.1.2.1 Off-the-Shelf Software Products

Off-the-shelf (OTS) software products are "software product(s) available for any user, at cost or not, and used without the need to conduct development activities" [21]. This definition of OTS indicates the main targeted user as the one discussed in Section 1.1.1.1, a nameless, faceless customer. The suppliers of OTS software face all quality-related dilemmas discussed previously, that is, cost, manifestation, knowledge, and time, and from what can be seen in the market, they do not deal with them too well. In their seventh decade, information technology companies still happen to deliver unreliable, poorly engineered, and sometimes surprisingly user-unfriendly products. Why?

Besides a short budget, undemanding customers, and lack of required knowledge, an OTS supplier is exposed to another challenge: a difficulty in communication with the users. A massive user is an unknown user, not reachable directly, and unfortunately also rather IT-ignorant, so not helpful in identifying missing or required quality. Then how exactly is the OTS developer supposed to build a product of required quality if he or she cannot talk to his or her customers?

There are several possible approaches to helping the developer create an OTS product of correct and appreciated quality. Some of them are:

- Collecting users' feedback through surveys
- Collecting detailed crash reports
- Sociological analysis of the targeted user groups
- Extrapolation.

In order to be effective, *collecting users' feedback* in the domain of quality requires a considerable effort of design. The questionnaire cannot be too long because the

responders will become disinterested, it cannot use specialized software quality vocabulary or concepts because it may be incomprehensible, it cannot be difficult in operation because it will discourage the user, but despite of all these constraints it has to bring the required information. Additionally, in the case of OTS developers that sell their product internationally the survey has to be *localized*, which means that it not only has to be properly translated but it also has to take into consideration the cultural context of the country in which it is being run. Another element that influences the usability of the survey is its statistical value. If any important decision about developing quality attributes and budget related to it is to be made, it cannot be based on partial or invalid information; in other words, it has to come from a statistically representative group of responders. If a software product is being sold in hundreds of thousands of copies, a few hundred replies to the survey will hardly constitute a statistically valid basis for any strategic decision.

Collecting crash reports seems to be a popular tool of getting real feedback, but sometimes its undisputable value is diminished by legal and financial reality. As the author began his adventure with IT technology in late 1970s, he has seen (and survived) hundreds, if not thousands, of different crash reports, blue screens, and event logs, trying in most cases to understand the information contained in them. The reports evolved from compressed, cryptic texts unavailable to the "uninitiated" to elaborate multi-page documents describing in almost-human language every detail of the crash. To appreciate the software quality engineering-related value of these reports it is important to stress something that may seem obvious: none of them contains the information of the type "the functionality Y is missing." Crash reports are almost purely quality-related data that should help the developers make a very good *next* version/update/build of their product. Where is the problem, then?

In order to be of any use, the reports have to be transmitted to the developers. The majority of applications use fully or partially crash report generation and transmission services of the operating system they reside upon, and these services are not free of charge. A considerable number of smaller developers never receive their reports because they simply cannot afford them.

Sociological analysis of the targeted user group is the tool that through dedicated research helps identify the most important needs of this group within a predefined domain. Applying it in software quality engineering brings information about customers' needs for such characteristics as usability (ease of use, learnability, etc.) or quality in use (productivity, effectiveness, satisfaction, etc.). A good, simple illustration of such an analysis process would be the project to develop a text processor. Before making any technological and financial decisions about the quality of the new product, the developer would have to ask the following questions:

- Who is the targeted user (e.g., a mass user or a specialist)? In what country or region? In what sector of the market?
- What would be main application areas of the processor for each of the categories of the user?
- What quality attributes are associated with every identified application area of the processor?

- Are all the attributes of the same weight or they can be prioritized? In how many and what priority levels?
- Which of these attributes are mandatory and which could be done later?
- What may happen in terms of product behavior if the mandatory attributes are absent?
- What would be foreseeable reactions of the targeted user to the lack of these attributes?

After having at least these few questions thoroughly answered, the developer may begin the decision process about design, technology, and budget for quality of the new product.

Extrapolation in terms of quality is an exercise, the objective of which is to identify successful quality attributes of the new (or being improved) product through observed reactions to existing and missing quality in products launched to date. Continuing the example of a text processor, it is quite possible to observe within, for example, five consecutive versions of the product the positive response to enhanced *operability* ("operability" measures the degree to which a product or system has attributes that make it easy to operate and control [22]). So, one of the important quality attributes in a new version would be observably increased operability. Of course, nothing like *global operability* would make sense, so the developer will have to identify what functions/functionalities/services would be preferred to have increased operability. In text processing, one of the most important functions is change tracking, but this particular function in some existing processors is uncomfortable in use, unclear, unintuitive, and so on, so its operability is considerably low. The *extrapolation* in this case would indicate that to attract more customers to their product, the developers should pay particular attention to change tracking function and made it considerably more user friendly.

1.1.2.2 On-Demand Systems

This category of systems and software comprises products that require a *user-specific intervention* from a developer prior to their installation. Such an intervention can go from a simple adaptation of an office support system (a small-sized system integration effort), through a dedicated configuration of existing "suites" (such as Oracle's "E-Business" [23] or SAP's "Business One" [24]) to a complete, from-scratch development of a required solution.

No matter the size and complexity of the developer's task, from a quality engineering perspective the basic conditions are the same:

- A user is known
- Requirements are identifiable
- Required expertise should exist
- Responsibility is direct.

A *known-user* situation should at least help open direct communication channels, which in turn should allow for a professional investigation of customer's real needs

of quality in the future system. As was stated in Section 1.1, the user's knowledge of quality engineering may be seriously limited, putting the majority of his or her justified quality requirements in the category of "unaware," hence the term *investigation*. Nonetheless, the developer deals here with relatively precise situation: a known user, a known or at least analyzable and definable problem, an identifiable required area of expertise, and available technology.

An *identifiable requirements* situation assumes that within an effort of creating an appropriate solution for the customer there are means to extract all relevant information necessary for further definition of correct and complete quality requirements. Keeping in mind that they may fall in all three categories differentiated by the level of difficulty in obtaining them ("stated, implied, unaware"), it can be understood that the process itself may be lengthy, demanding several iterations and a particular effort in presenting, explaining, and justifying the identified requirements. And in case of systems developed on demand, this is a sole responsibility of the developer. For more details, go to Section 2.2.1.

Required expertise simply means all expertise necessary for making quality happen in a developed system (discussed in detail in Chapter 2). What is important in real-world development situations is the *existence* of this expertise. A very popular and equally incorrect perception of quality limits it to the equivalent of "tests" or "no bugs." It is obvious that the crashless behavior of given software improves its use and increases the positive reception by a customer, but from the perspective of an overall quality, tests make only a part of the required expertise. To prove it is enough to analyze any domain-recognized software or system quality model, such as, for example, the most recent ISO/IEC 25010 [25] or even classical Boehm model [26, 27]. From this analysis it can be found that even (ideally) a bugless system may receive a low grade on quality because its productivity is not what was expected, or its use, be it business- or maintenance-related, is difficult, so slowing down the work and tiring the user. Further in this direction, the content of such a model brings the real structure of expertise required in quality engineering. If the model from ISO/IEC 25010 is taken as the reference, the required expertise spans from applying quality to architecture, design or coding, software measurement, and security mechanisms (internal/static quality), through operation and maintenance and all their related quality characteristics and attributes (external/dynamic quality), up to productivity, psychometrics, and sometimes even psychology (usability, quality in use). And to that list, the ability to design, plan, and execute required measurements and evaluate the results has to be added. It is understandable then that in industrial reality, the full coverage of such an experience would be difficult to come by, but what seems to be a real problem is that this expertise in general is too *scarce*.

One of the most important elements of on-demand development is the *direct responsibility* of the developer to his or her customer. Together with a properly constructed contract, it gives to both the customer and the supplier the tools to demand and obtain (or pursue, if need be), even if the demands are not exactly of the same nature. On one hand the imperfections in requested quality of the delivered system can be directly traced back to the supplier, properly proven and reacted upon in an appropriate legal, financial, or technical way. On the other hand, if the reasons

for such a situation are rooted in, for instance, lack of cooperation from the customer, it can also be proven helping the supplier even the odds.

1.2 COST OF QUALITY

In the following chapters, the cost of quality will be discussed from two different perspectives: how to position the costs of engineering quality into software or a system in the overall project budget, and how much the consequences of missing quality may cost.

The first perspective (Section 1.2.1) analyzes the financial ramifications and challenges that the process of engineering quality into the largely understood information technology domain faces in the real, industrial world.

The second perspective (Section 1.2.2) attempts to answer a very important question: what might happen if the quality is not there?

1.2.1 Economic Ramifications of Software Quality Engineering

When undertaking the challenge of engineering quality into software, one could take into consideration a few basic facts from life:

- Everything in software engineering boils down to the user's satisfaction
- Satisfaction is conditional to the overall behavior of the system, with software products in the first place
- The behavior of any software product is perceived through features and quality
- Features and quality of software product are expressed through requirements
- Any behavior-related requirement for software product may only be realized through code.

Having these points in mind, let us open the discussion about financial ramifications of engineering quality into software or system with the following statement: *In most development projects, functionality and quality are natural enemies.*

Is this really true? Unfortunately for all IT users, yes. There are in fact very rare situations where the project budget is open; in all other cases, the budget defines the battlefield where functionality and quality fight for an upper hand (Fig. 1.2).

As shown in Fig. 1.3, function–quality–cost (FQC) economic perspectives are merciless: no matter how big the budget is, there always will be competition between features and quality.

It translates into a financially valid fact illustrated in Fig. 1.4, implementing features and quality costs, so for a constant budget (C) more features (af) means less quality (bq). And the opposite is also true, however it is much more rare.

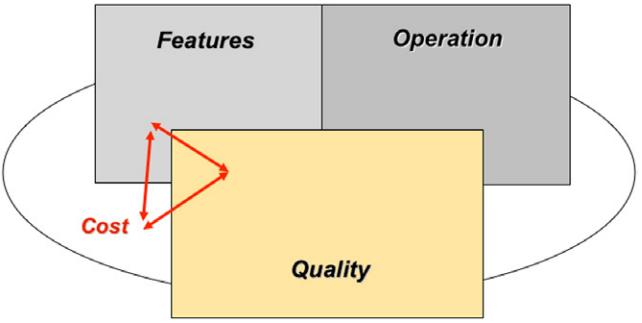

Figure 1.2 Functionality–quality battlefield.

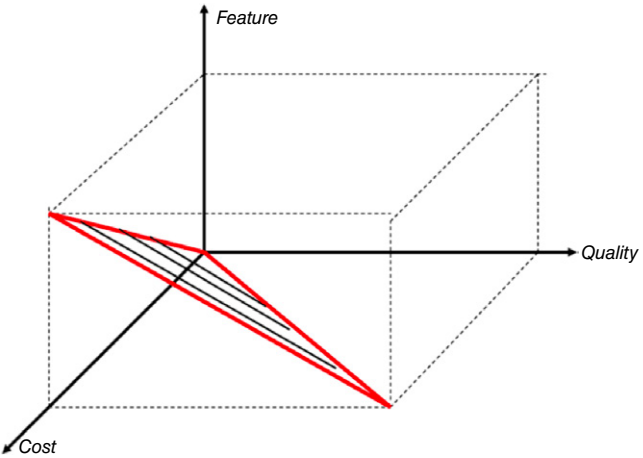

Figure 1.3 Economic perspective of implementing features versus quality (FQC).

Economic Perspective

Cost = aΣ features + bΣ quality aspects

or

Cost = af + bq

Figure 1.4 Theoretical model of financial competition between features and quality (FQC). a, b = investment levels; f = features; q = quality aspects.

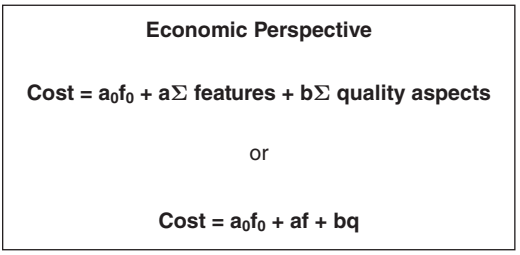

Figure 1.5 Corrected model of financial competition between features and quality (FQC). a_0 = initial investment level; f_0 = initial set of features; a, b = investment levels; f = features; q = quality aspects.

The analysis of the model presented in Fig. 1.4 will immediately show that such a model, even if mathematically correct, is in fact purely theoretical. One can imagine a software product that will have features (their quantity is of no importance here) associated with the appropriate level of investment ($af \neq 0$), but being developed with no regard to quality ($bq = 0$). It is however much more difficult to imagine a product having no features ($af = 0$), but exhibiting certain quality ($bq \neq 0$). To correct this unrealistic representation the model has to take into account the fact that a software product that does not have at least a minimal, initial set of features does not exist. In the corrected model (Fig. 1.5), this initial set of features is represented by a_0f_0.

It is now easy to understand why in projects of a *predefined* budget, quality and functionality are enemies. And it is even easier to foresee the winner. From what can be observed in the market of software products, features continuously win, even if such victories quite often prove short-sighted. The first positive impressions based on functional richness quickly turn into disappointment or rage when the software starts producing "blue screens."

So is a software quality engineer on a by-default-lost position? Well, such a position surely is not an easy or a comfortable one, but it is still manageable and gives chances of success, if only some thoughts from the following were taken into consideration:

- From the very beginning, negotiate functional requirements with quality requirements in mind. "Later" may be too late!
- Evaluate the list of features against the budget as soon as possible. This will be your first indication about a level of possible quality, and your first argument in renegotiating the FQC proportions.
- Any functionality has its quality counterpart. Find it!
- The quality counterpart may require development or any other form of "expenditures." Take it into account when evaluating the project.
- Analyze well the existing FQC. If the quality part is considerably low, the project may quickly run into a high-risk scenario.

- A new functionality may kill the overall quality of the product, so negotiate carefully.
- A new quality requirement rarely or never harms the product.

The economic ramifications discussed in this chapter represent the point of view related to a development process and effort and as such can be considered as *internal*. The *external* ramifications attempt to analyze financial aspects of engineering quality into software in its broader, social context, also known as the *cost of missing quality*.

1.2.2 Cost of Missing Quality

The fact that IT systems are essential for the majority of tasks in human society raises a question, very important to both IT users and IT suppliers: What are the consequences of missing quality of an IT system in active use?

Every system has to make compromises in several areas and quality attributes are no exception. Different systems are subjected to different risks as they have specific quality attributes, which usually are different from one system to the other. In the ideal world, every quality attribute would be at the highest level for every system, but in practice this is not possible. As the application areas of IT systems are diversified, decisions must be made regarding which quality attributes should be given what priority in terms of the possible impacts to this area. Also, for the same reason, the cost of missing quality is different from one application area to another.

To analyze the costs of missing quality, the first helping step is to categorize the IT system in question, as within every category there are quality attributes specific or "most valuable" to it. In real-life cases, such a basic analysis should be but the beginning of a much more exhaustive process, where an impact of the absence of each application area-related quality attribute of the system has to be identified and evaluated.

The objective of the evaluation is to demonstrate the consequences of missing quality to the decision makers within an organization and, by doing so, to help them make the correct technical and budgetary decisions and prioritize the quality attributes for a system.

1.2.2.1 Cost Analysis-Based Approach

The missing quality cost (MQC) is translated into an impact on people and organizations, relative to the operation domain of the IT system. In this chapter, the MQC is analyzed applying Eppler and Helfert principles [28] with costs classified in two categories: direct and indirect.

Direct costs are directly linked to missing quality. They consist of the effects that are observable immediately after unfortunate events happen. Examples of direct costs are:

- Compensation for damages
- Physical injury and related compensations.

Indirect costs are difficult to calculate, as they may not be visibly linked to missing quality. Consequently, it is often difficult to identify them and they may remain hidden for a long time or even never discovered. Some examples of indirect costs are:

- Lost reputation or market position
- Wrong decisions or actions
- Lost investment.

1.2.2.2 Impact Analysis-Based Approach

Missing an essential quality attribute in software usually costs both the customer and the supplier, however not necessarily in equal proportions. The customer can lose data or his or her business or even, in the worst case, be exposed to physical injuries to the extent of death. Other, less dramatic impacts may include the costs linked to technical support and the costs of wasted time in investigating the source of problems. In addition, the customer may also lose his or her credibility if, due to too-low quality of his or her IT system, he or she cannot meet his or her commitments toward customers.

The cost to the supplier is most often of a different nature than the cost to the client, but there are significant impacts as well. For example, the costs of technical support can be very high due to the number of clients requesting it. Other costs include handling a large number of customer complaints, development costs to fix bugs, and the costs of supporting multiple version of the same product. Finally, the supplier may be also be pursued by the law, or forced to pay penalties to the limits of bankruptcy or loss of the market.

1.2.2.3 Risk Analysis-Based Approach

Risk analysis is an essential tool in determining the MQC, as the cost itself is usually linked to an event that could (or should) happen as the consequence of missing quality. Moreover, as the place and time of the events related to missing quality may sometimes be difficult to determine, one of the better methods for evaluation of the cost of missing quality is the classical risk analysis approach.

The risk is characterized by its probability p (where $0 < \mathrm{p} < 1$), and impact L, also known as the potential loss (where L represents a quantity in measurable units, such as currency) [29].

Risk exposure (*RE*) is the product of the risk probability and its potential loss. This simple approach is further used when individual categories of IT systems are analyzed:

$$\mathrm{RE} = \mathrm{p} * \mathrm{L}$$

Both probability p and impact L are strongly related to the level of criticality of the analyzed IT system. The most broadly known scale of the criticality in IT domain is the standardized IT system criticality levels schema published in the IEEE Standard for Software Verification and Validation [30]. The levels are:

Level A (Catastrophic)

- Continuous usage (24 hours per day)
- Irreversible environmental damages
- Loss of human lives
- Disastrous economic or social impact.

Level B (Critical)

- Continuous usage (version change interruptions)
- Environmental damages
- Serious threats to human lives
- Permanent injury or severe illness
- Important economic or social impact.

Level C (Marginal)

- Continuous usage with fix interruption periods
- Property damages
- Minor injury or illness
- Significant economic or social impact.

Level D (Negligible)

- Time-to-time usage
- Low property damages
- No risks on human lives
- Negligible economic or social impact.

1.2.2.4 Example

To illustrate the process of analyzing the consequences of missing quality, the following context based on real events (described in [31]) will be used:

- IT system application area: Nuclear power plant, system monitoring and synchronizing chemical and diagnostic data from primary (nuclear reactor) control systems.
- Quality subcharacteristic: *Recoverability* from reliability quality characteristic of ISO/IEC 25010 quality model [25].

Let's further imagine that the objective of this analysis is to convince the decision makers that much more money has to be invested into quality in general and *recoverability* in particular.

The process in simple steps could go in the following manner:

Step 1: Identify the system behavior related to the targeted quality subcharacteristic or attribute. *Recoverability* represents the level of the ability of the system

to correctly recover from a serious disruption (be it a crash, an unscheduled shutdown, or even a not entirely successful update).

Step 2: Identify the criticality level of the system. The important question in this step would be: What may happen when the system that monitors and synchronizes sensitive chemical and diagnostic data from reactors recovers incorrectly? This question invokes a few more detailed questions, such as:

- What can be lost?
- What can be corrupted?
- What may happen if data or system states are corrupted (wrong)?
- What may happen if data or system states are lost?

It would be prudent to answer these questions applying the method of the worst-case scenario. In the case of the real events described in Reference 31, the corrupted (reset) data forced "safety systems to errantly interpret the lack of data as a drop in water reservoirs that cool the plant's radioactive nuclear fuel rods. As a result, automated safety systems at the plant triggered a shutdown." And this outcome can be considered very positive. In the worst-case scenario, the automated safety systems could interpret the wrong data in the opposite way (as a water overflow) and let the rods eventually melt down, causing a real disaster.

So what would be the criticality level of the analyzed system? It is not a system that directly controls the reactors but it should exhibit continuous usage capacity (with only version change interruptions), and it surely can invoke environmental damages, create serious threats to human lives, or important economic or social impact. So perhaps Level B? But what if the previously mentioned worst-case scenario should happen? The high level *recoverability* of this system could help avoid eventual further negative consequences leading to a disaster by not sending the confusing data to systems that directly control the reactors. So perhaps Level A? This decision may be taken either from the perspective of required financial efforts (so most probably Level B) or social and environmental consciousness and responsibility (so Level A), but whatever it will be, it requires a solid justification.

Step 3: Risk analysis. In this step the probability p, impact L, and risk exposure RE should be estimated in order to create information required in Step 4, cost estimation. The probability of the occurrence of the negative events related to a low level of system recoverability can be obtained through active measurements, accumulated historical data, or even observed trends in the system's behavior. In an ideal situation, the analyzed system would be disconnected from active operation and undergo a series of experiments with controlled disruptions and measured outcomes. The probability p would be calculated as the ratio between the number of experiments that created corrupted data after the recovery and the number of all experiments. Of course, information obtained in such a way would be coarse, as not every corruption of data would automatically lead to melting of radioactive rods, but it would be a solid indicator nonetheless. In real life, such an indicator can be obtained by monitoring the system behavior over a given time period and calculating the ratio between disruptions that ended with corrupted data and all disruptions that took

place. To perform a precise risk analysis the probability p should be, however, calculated separately for each important category of impacts, such as what percentage of data corruption after recovery would provoke an event of false cooling water overflow indication.

Impact L has partially been analyzed in Step 2. In Step 3, the set of most important impacts should be chosen and linked to their respective probabilities. To calculate the risk exposure *RE*, both global and individual per impact, each impact should be translated into its mathematical representation, in its most trivial form, money. Then a simple multiplication $p \times L$ will give the values of risk exposure *RE* necessary for the cost analysis performed in Step 4.

Step 4: Cost analysis. The *RE* values obtained in Step 3 are just dry numbers that do not represent the totality of costs associated with the absence of an identified quality attribute. They may be interpreted as direct or immediate costs but the full cost analysis has to take into consideration also indirect costs, nonmonetary costs, the risk context, and, last but not least, the cost of required improvements/modifications of the system that would remedy the problem. To better explain this notion, let's take the following hypothesis: the impact of melting the rods in one of the nuclear reactors would be a (sure) destruction of the environment in the radius of 50 miles for next 70 years, a (probable) loss of human lives, and a (sure) economical disaster to the surrounding community, but the probability of it all happening as a cause of low recoverability of the system is a small but firm 1.5%. At the same time, remedying the problem would require a considerable investment (quite often the case where the legacy systems are mixed with newer generation ones). Even if everything from the preceding list capable of being transformed into monetary value was transformed so, the resulting *RE* would be probably relatively low, plus an extra investment required to better the existing system as the counterargument; but should it be ignored?

Step 5: Convincing the decision makers. Imagine the following exposé of yours:

> Ladies and Gentlemen,
>
> The recent analysis of our system monitoring and synchronizing chemical and diagnostic data from primary (nuclear reactor) control systems shows that its quality, in particular its recoverability, is insufficient and requires immediate intervention.
>
> This intervention will require $\$X$ of investment and Y months of work of our (our supplier's) IT team.
>
> The following are the data: during last N months the system went into the recovery state M times with (for example) 30% of occurrences of corrupted, after-recovery data. (For example) 1.5% of these occurrences are related to the reactor core cooling water control. We estimated the impact of possible overheating of the rods as a (sure) destruction of environment in the radius of 50 miles for next 70 years, a (probable) loss of human lives, and a (sure) economical disaster to the community around. The rough estimation of *RE* is \$R but the overall cost, should this disaster happen, is much greater for the community, for the environment, and for our organization (insert here the list of nonfinancial consequences).

Taking this information into consideration, please grant the resources required to improve the actual situation.

The above exposé is just an example or even a template that can easily be reused in most *negative-motivation* cases of "what we lose if we do not do it" type.

Another option is a *positive-motivation* approach, or "what we gain if we do it" philosophy. The general methodology is the same, but instead of counting the possible losses, the process focuses on gains that the addition of a missing quality attribute or improvement of an existing one may bring to the system and in consequence to its creators, its users, and possibly to environment.

1.2.3 Some Important Quality Characteristics of Chosen Categories of IT Systems

In the report published in Reference 32, the authors proposed the taxonomy of most popular IT systems distributing them into four categories and eight subcategories (see Table 1.1). The discussion presented further in this chapter is based on this taxonomy.

1.2.3.1 Decision Support Systems

The main goal of decision support systems, as their name implies, is to help organizations and individuals in the process of decision making. Decision support systems usually combine data from different sources with sets of rules for analyzing them and, like all software, are subject to a set of common risks associated with the nature of software, but also possess several challenges of their own. A considerable percentage of decision support systems depend on external data sources, hence they are particularly sensitive to the quality of the data they receive to process. Another important issue that the decision support systems face are incorrect analysis algorithms.

In consequence, the important quality subcharacteristics found representative for decision support systems are *accuracy*, *analyzability*, and *suitability* [32]. These

Table 1.1 IT Systems Taxonomy

Information System Categories	Information System Subcategories
Transaction processing systems	Transactional applications systems
	Financial applications systems
Computer-based communication systems	Telecommunication
	Network management
Management information systems	Management information systems
	Information management systems
Expert systems	Decision support systems
	Industrial support (control) systems

three subcharacteristics may constitute the starting point for further analysis of quality required in a particular realization of a decision support system.

1.2.3.2 Industrial Support (Control) Systems

Industrial support (control) systems (ICSs) collect and process information related to industrial processes. A typical ICS consists of a series of sensors monitoring an industrial process and a software system to process the received data and make decisions required to properly execute the controlled process. In practice, most contemporary industrial processes use some kind of ICS. This category of systems varies from small and simple ones controlling noncritical processes to large and complex systems overseeing and running whole plants. The latter are particularly exposed to very high impacts if their ICSs do not perform as expected. Depending on the nature of the system, the impacts can be as great as irreversible damages to the environment, loss of human lives, and very high financial losses, thus, in general, it would be recommended to classify these systems at Level A of the scale presented in Section 1.2.2.

In consequence, the starting point for full quality analysis would be the quality subcharacteristics of *testability*, *accuracy*, *fault tolerance*, and *adaptability* [32].

1.2.3.3 Transaction Application System

By definition, a transaction is an individual and indivisible operation that in order to be considered completed has to be executed in its entirety. This condition is closely linked to the mechanism of *rollback*, the role of which is to get both ends of the transaction to its initial state, should the transaction fail. The most broadly known type of transaction processing is banking, where, for example, a transfer of funds from one account to the other is considered successful only when the recipient's account sends the confirmation and the sender's account receives it. In all other cases, rollback should secure the reliability of the transaction itself and force both accounts to their state from before the transaction. In more general terms, the transaction application system category consists of the systems that process information in a transactional way, ensuring that any transaction performed by them is completed or cancelled successfully. These systems also allow multiple users to manipulate the same data, usually distributed so their consistency is also of highest importance.

As the research in Reference 32 shows, the important quality characteristics found representative for transaction application systems are *functionality*, *reliability*, *usability*, and *efficiency*.

1.2.3.4 Financial Transaction Systems

A popular description of financial transaction that can be found on one of many open fora would be: "It is an event or condition under the contract between a buyer and a seller to exchange an asset for payment. In accounting, it is recognized by an entry in the books of account. It involves a change in the status of the finances of two or more businesses or individuals" [33].

The main goal of a financial transactions system is to automate the handling of financial operations. Some most popular examples of this type of systems are

purchase applications, loans management systems, mortgage management systems, systems to manage bank accounts, systems to manage credit card purchases, and systems to manage debit card purchases. In all cases, quality attributes (or subcharacteristics) of accuracy, maturity, and recoverability seem to be essential. These subcharacteristics can be further folded into two main quality characteristics for the financial transaction systems: *functionality* and *reliability* [32].

1.2.3.5 Network Management Systems

Network management systems manage, administer, and monitor networks on which organizations rely to carry data from node to node. These systems have to be interoperable, reliable, and tolerant to faults, as most of their users cannot afford to have communications seriously disrupted [34]. According to the research presented in Reference 32, the most important quality factors for this type of systems are *fault tolerance*, *interoperability*, and *operability*.

1.2.3.6 Telecommunication Systems

Telecommunication systems are the backbone of the telecom operator's business model. They use huge infrastructures such as telecommunication towers, satellites, and undersea cables and regroup the operation, administration, maintenance, and provisioning functions. These management functions executed by large IT structures provide systems or networks with fault indication, performance monitoring, security management, diagnostic functions on traffic, configuration, billing, and user data provisioning. What has also to be taken into account is the fact that the existing telecommunication technology varies from older systems embedded in various types of hardware to modern, fully *soft* installations, and all of them have to cooperate and coexist in a productive manner. Quality characteristics that address these concerns would be: *functionality*, *reliability*, *usability*, and *efficiency* [32].

1.2.3.7 Management Information Systems

Management information systems are primarily used by managers and business domain experts to make business forecasts and decisions. As these users usually have a limited IT proficiency, the ease of use is a key efficiency feature. From a business use perspective, management information systems provide data necessary for strategic decision making, with services such as:

- Generating financial statements, as well as inventory reports or sales status reports.
- Answering managers' questions by offering different decision scenarios with their results.
- Supporting human resources-related decision making.
- Providing information for analysis and budget planning.
- Facilitating audits by giving complete audit trail.

The quality subcharacteristics important for these services are: *functionality*, *usability*, and *maintainability* [32].

1.2.3.8 Information Management Systems

"Information management system" is a broad term that describes a multitude of systems of which the main objective is to manage information. Some of the subcategories of these types of systems are content management systems, document management systems, digital asset management systems, or geographic information systems [35].

The common functionality of such systems is the ability to retrieve, store, and manipulate information. What is interesting is the fact that in many cases the critical risk factor that affects these systems is not related to the system infrastructure itself, but to the information they manage. In consequence, as found in Reference 32, the most important quality factors for this type of systems could be considered *security*, *operability*, *accuracy*, and *changeability*.

1.2.3.9 Practical Observations

One of interesting observations made in the course of analyzing the relationship between the category of an IT system and quality characteristics important for its use was the finding that missing "crown" quality attributes that characterize this category are not always the ones that make it fail. As some case studies have showed (see the Appendix, Case 11), the lack of a quality attribute off the main list (obviously wrongly seen as "minor") sometimes may have a bigger impact than one considered "main." This is why the idea discussed already in Section 1.2.2 will be repeated: In real-life cases, the analysis of typical-for-the-system quality attributes should be but the beginning of a much more exhaustive process, where an impact of the absence of each application area-related quality attribute of the system has to be identified and evaluated.

And finally, ensuring quality in software enhances operational effectiveness and helps accomplish strategic objectives of the organization:

- Developing modern, reliable, and environment-friendly solutions
- Keeping costs and spending low
- Keeping customers and adding new ones by giving a good service that meets and exceeds expectations.

If these goals are to be effectively achieved, software quality must make significant progress in terms of its recognition and importance in the business world, where the costs associated with missing quality should be treated in more explicit, prominent, and measurable ways.

1.3 QUALITY OF A SOFTWARE PRODUCT AS AN INDICATOR OF MATURITY

Is quality really an indicator of maturity? It is not an ultimate and always true evaluation, but in most cases quality goes with maturity. Young, immature companies

usually cannot afford developing more than just a set of attractive functionalities, whereas mature organizations can develop quality too, so in this sense the level of quality observed in a software product is an indicator of the level of maturity of its developer. When evaluating the maturity of a software development organization, one can apply sophisticated methods and models such as CCMI, SPICE, or ISO 9000 and still arrive at conclusions that may not entirely reflect the reality. All the best processes will not replace the tangible indicators of the real maturity: functionalities and quality of the product. One may even say that because functionalities are always in a product and quality is only sometimes present, quality is a more restrictive indicator.

1.3.1 CMM/CMMI

The Capability Maturity Model (CMM) was born in 1990 as result of the research effort conducted by specialists from Software Engineering Institute (SEI) of Carnegie Mellon University [7]. Its next version, Capability Maturity Model Integration (CMMI^{SM}), is known in the industry as a best practices model. It combines practices of systems engineering (SE), software engineering (SWE), integrated process and product development (IPPD), and supplier sourcing (SS) disciplines. The CMMI is mostly used to "provide guidance for an organization to improve its processes and ability to manage development, acquisition, and maintenance of products and services." The CMMI (Table 1.2) was conceived to allow organizations to rely on a single model to evaluate their maturity and process capability, establish priorities for improvements, and help them improve their practices.

The CMMI is available for various combinations of disciplines in two representations: "staged" and "continuous." The model is divided into process areas (PA), each of which contains a set of generic and specific practices (Fig. 1.6) that through their existence (or lack) may manifest the maturity of an organization. In search for references to quality in CMM/CMMI manual one immediately finds the following announcement: "[In CMM] the phrase 'quality and process-performance objectives' covers objectives and requirements for product quality, service quality, and process performance."

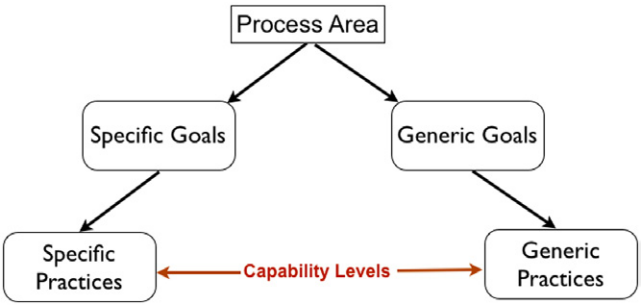

Figure 1.6 CMMI model components (adapted from [7]).

Table 1.2 Capability Maturity Model of SEI (adapted from [7])

Maturity Levels	Process Areas
5. Optimizing	Causal analysis and resolution
	Organizational innovation and deployment
4. Quantitatively managed	Quantitative project management
	Organizational process performance
3. Defined	Organizational environment for integration
	Decision analysis and resolution
	Integrated supplier management
	Integrated teaming
	Risk management
	Integrated project management for IPPD
	Organizational training
	Organizational process definition
	Organizational Process Focus
	Validation
	Verification
	Product Integration
	Technical Solution
	Requirements Development
2. Managed	Configuration management
	Process and product quality assurance
	Measurement and analysis
	Supplier agreement management
	Project monitoring and control
	Project planning
	Requirements management
1. Initial	None

More detailed analysis will yield more than 400 references to "quality" within the CMM/CMMI manual, but all of them will bear the notion of *a process* that in one way or another should *help* create a software product of quality. A quick illustration of the presence of the subject of quality within the maturity levels could look as follows:

- Level 1: None
- Level 2: Specific objectives for the performance of the process (e.g., quality, time scale, cycle time, and resource usage)
- Level 3: Same as Level 2
- Level 4: The quality and process performance are understood in statistical terms and are managed throughout the life of the process

- Level 5: Select and systematically deploy process and technology improvements that contribute to meeting established quality and process-performance objectives.

What is then the link between the maturity of an organization and quality of its products? First and foremost: it is *nonautomatic*. The organization may have all best processes in place, be continuously certified ISO 9000, and still manufacture products that will not survive a day. The level of maturity could be compared to the knowledge of a battlefield—the deeper that knowledge is, the higher are the chances of victory. But they are still only chances, not certainty.

1.3.2 SPICE ISO 15504

Software Process Improvement and Capability Determination (SPICE) is an international initiative to support the development of an International Standard for Software Process Assessment [36]. The first working draft was developed in June 1995, with the release to ISO/IEC for the normal process for development of international standards. In 1998 the documents were published as ISO/IEC TR 15504:1998—Software Process Assessment. As of now, SPICE in its ISO/IEC 15504 international standard form has ten parts, the publishing of which spans over the last decade:

- Part 1: Concepts and vocabulary
- Part 2: Performing an assessment
- Part 3: Guidance on performing an assessment
- Part 4: Guidance on use for process improvement and process capability determination
- Part 5: An exemplar process assessment model
- Part 6: An exemplar system life cycle process assessment model
- Part 7: Assessment of organizational maturity
- Part 8: An exemplar process assessment model for IT service management
- Part 9: Target process profiles
- Part 10: Safety extension

SPICE, or ISO/IEC 15504 series of standards, provides a framework for the assessment of processes. This framework can be used by organizations involved in planning, managing, monitoring, controlling, and improving the acquisition, supply, development, operation, evolution, and support of products/services. Process assessment examines the processes used by an organization to determine whether they are effective in achieving their goals. The results may be used to drive process improvement activities or process capability determination by analyzing the results in the context of the organization's business needs, identifying strengths, weaknesses, and risks inherent in the processes.

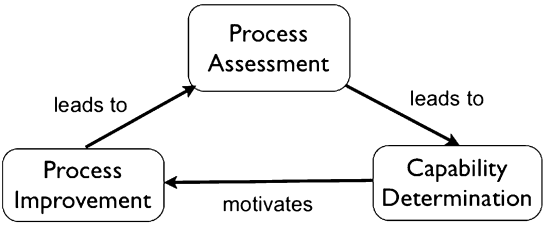

Figure 1.7 Process assessment relationship (adapted from [36]).

SPICE provides a structured approach for the assessment of processes for the following purposes:

- By or on behalf of an organization with the objective of understanding the state of its own processes for process improvement
- By or on behalf of an organization with the objective of determining the suitability of its own processes for a particular requirement or class of requirements
- By or on behalf of one organization with the objective of determining the suitability of another organization's processes for a particular contract or class of contracts.

The framework for process assessment proposed in SPICE is intended to facilitate self-assessment, provide a basis for use in process improvement and capability determination, take into account the context in which the assessed process is implemented, produce a process rating, address the ability of the process to achieve its purpose, be used across all application domains and sizes of organization, and give the chance for an objective benchmark between organizations.

Through this, the organization is expected to become a capable organization that maximizes its responsiveness to customer and market requirements, minimizes the full life cycle costs of its products, and as a result maximizes end-user satisfaction.

As can be seen in Fig. 1.7, SPICE has two principal contexts for its use: process improvement and capability determination. The relationship between SPICE, process maturity, and software product quality is indirect and bears features such as these indicated when CMM/CMMI was discussed. Maturity and efficiency of processes existing in an organization that develops software make without doubt very important foundations for the quality of product, but here the influence ends. The rest must be done by software engineers who know how to put quality into what they are about to produce.

1.3.3 SWEBOK

The purpose of the Guide to Software Engineering Body of Knowledge, called further SWEBOK, is "to provide a consensually-validated characterization of the

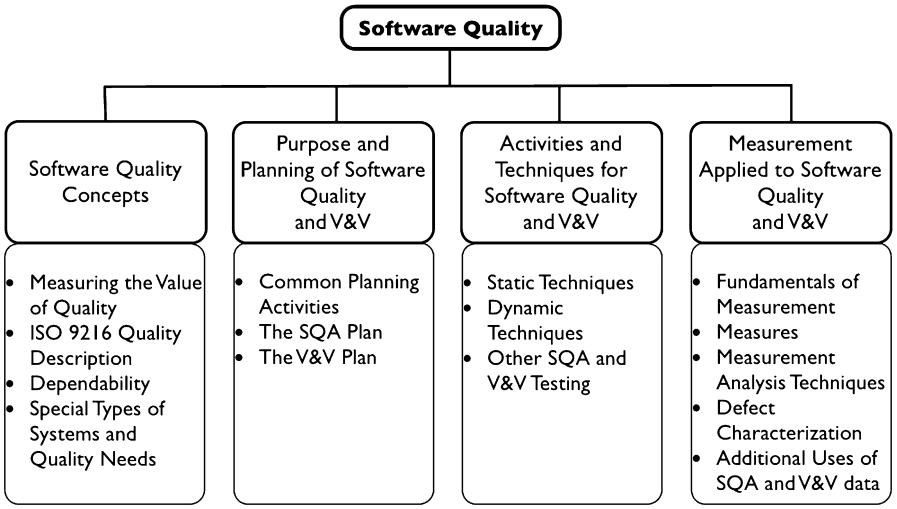

Figure 1.8 Breakdown of topics for software quality (adapted from [2]).

bounds of the software engineering discipline and to provide a topical access to the Body of Knowledge supporting that discipline." To address this objective, the 2004 edition of Body of Knowledge is subdivided into ten knowledge areas (KA) and "the descriptions of the KAs are designed to discriminate among the various important concepts, permitting readers to find their way quickly to subjects of interest" [2].

Among these ten knowledge areas, a KA dedicated to software quality has its distinctive place. Like all other KAs within SWEBOK, the software quality subject is broken down and then discussed in individual topics (15) (Fig. 1.8) grouped in four sections:

- Software Quality Concepts (SQC)
- Purpose and Planning of SQA and V&V (P&P)
- Activities and techniques for SQA and V&V (A&T)
- Other SQA and V&V Testing (OT).

As the content of the Software Quality KA of SWEBOK is rather voluminous, it cannot be discussed to its full extent in this chapter, however, some "reader's digest" given below could help the reader identify the subjects of his or her particular interest and then further pursue them through the lecture of the full text of the guide.

In *Software Quality Concepts*, the guide discusses the issues linked to identification and management of costs related to quality (and indirectly to its lack) and modeling of quality, stresses the importance of quality in the context of dependability of software products, and points out the existence of quality perspectives other than these "classical" perspectives perceived through the lenses of ISO/IEC 9126 series of standards [37 to 40].

Purpose and Planning of SQA and V&V analyzes planning and objectives of software quality assurance (SQA) and verification and validation (V&V) processes in the context of what, when, and how quality should be achieved.

Activities and Techniques for SQA and V&V tackles practicalities of SQA and V&V execution, presenting among the others static and dynamic techniques recommended for these processes.

Measurement Applied to SQA and V&V presents basic notions of measurement theory and practice in context of software and software quality measurement.

As profound as may be the way in which SWEBOK discusses software quality, it still leaves some room for additional perspectives. One of them is the *engineering* perspective of making real quality happen (or, using simpler vocabulary, hands-on engineering interventions). This hypothesis lies at foundations of the research program conducted and published in 2006 [41] with the objective of evaluating each KA constituting SWEBOK in order to verify the level of representation of the subject of software quality engineering in this most prominent document of software engineering domain.

As part of this research, the latest version of SWEBOK had been analyzed from the perspective of the core processes constituting the practices of software quality engineering. These core processes were identified through the analysis of software life cycle processes published in the standard ISO/IEC 12207: 1995 (the 2008 version was still in redaction) [42] supported by the results of the research on software engineering principles [43] and software quality implementation models (discussed further in Section 2.3.1).

The dedicated analysis methodology developed in order to execute the research program consisted of four phases, presented Fig. 1.9:

- *Phase 1:* Analysis, validation, and, if necessary, addition of definitions of software quality engineering processes in Software Life Cycle (SLC) processes and activities identified in ISO/IEC 12207.
- *Phase 2:* Analysis, validation, adjustment (if necessary), and mapping of the set of basic processes of software engineering definitions identified in ISO/IEC 12207 with the processes and activities described in respective KAs of SWEBOK.
- *Phase 3:* Application and assessment of results of the mapping between the processes of software quality engineering identified in Phase 1 with the processes and activities described in KAs of SWEBOK (Phase 2).
- *Phase 4:* Identification and definition of applicable modifications to SWEBOK.

The results published in [41] take several pages of tables and definitions; however, some general conclusions that emerged from this research would be:

- Quality as engineering process is addressed in a limited form, to say the least
- The basic quality engineering activities such as quality requirements specification or modeling are not recognized anywhere

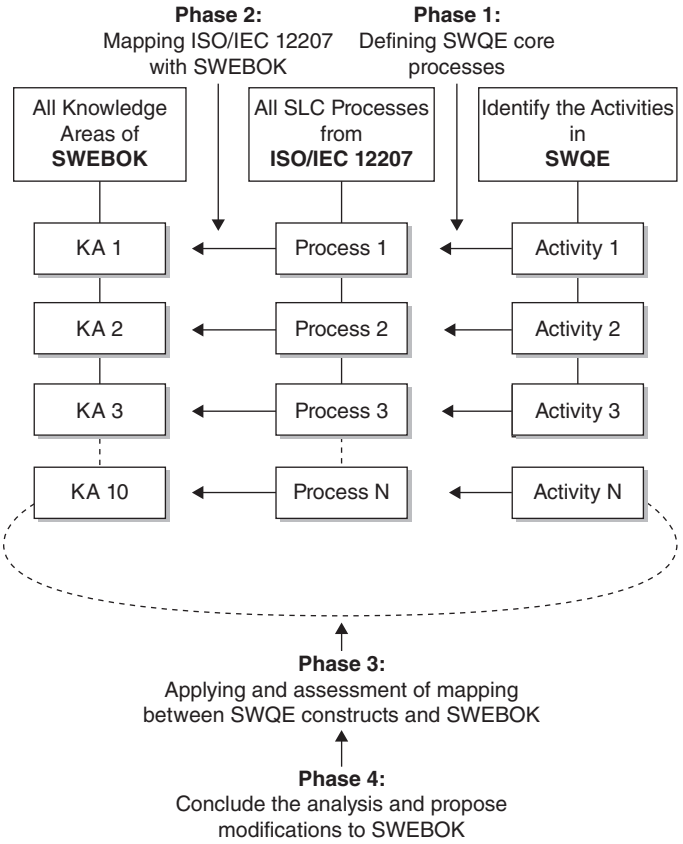

Figure 1.9 The architecture of the research methodology software quality engineering in SWEBOK [41].

- Quality testing is discussed almost only in reference to V&V processes, while in fact real evaluation of software product happens all along the life cycle
- Practical aspects of engineering quality into a software product are entirely omitted, while their appearance would be helpful at least in Software Construction KA.

Apart from a very basic conclusion about modifications that could enrich SWEBOK, one other of a more general nature comes to mind: software quality engineering, like its mother domain, software engineering, is still far from gaining stability and maturity and requires continuous research effort supported by wide cooperation with the IT industry.

The development works on SWEBOK are continued as a joint effort between the ISO/IEC JTC SC7 committee and the IEEE Computer Society, giving as the result several enhancements to its 2004 edition, including new KAs. The new edition is expected to be publicly available in 2014.

REFERENCES

1. École de technologie supérieure, education program in software quality. Montreal, Canada. Available at http://www.etsmtl.ca.
2. Abran A, Moore JW, Bourque P, Dupuis R, editors. *Guide to the Software Engineering Body of Knowledge*, 2004. Los Alamitos: IEEE Computer Society, 2004.
3. ISO 9000 Quality Management Systems—Fundamentals and Vocabulary. Geneva, Switzerland: International Organization for Standardization, 2005.
4. Highsmith J. *Agile Software Development Ecosystems*. Addison Wesley, 2002.
5. Kitchenham B, Pfleeger SL. "Software Quality: The Elusive Target." *IEEE Software* 1996; 13(1):12–21.
6. ISO 9001 Quality Management Systems—Requirements. Geneva, Switzerland: International Organization for Standardization, 2008.
7. CMMI-SE/SW/IPPD/SS 2002. CMMI Team, Capability Maturity Model Integration for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS), Version 1.1, Continuous Representation. Pittsburgh: Software Engineering Institute, Carnegie Mellon University, 2002.
8. Voas J. "Assuring Software Quality Assurance." *IEEE Software* 2003; 20(3):48–49.
9. Dromey RG. "Cornering the Chimera." *IEEE Software* 1996; 13(1):33–43.
10. Haley TJ. "Software Process Improvement at Raytheon." *IEEE Software* 1996; 13(6):33–41.
11. Diaz M, Sligo J. "How Software Process Improvement Helped Motorola." *IEEE Software* 1997; 17(5):75–81.
12. Georgiadou E. "Software Process and Product Improvement: A Historical Perspective." *International Journal of Cybernetics* 2003; 1(1):172–197.
13. Laitinen M. "Scaling Down Is Hard to Do." *IEEE Software* 2000; 17(5):78–80.
14. Boddie J. "Do We Ever Really Scale Down?" *IEEE Software* 2000; 17(5):79–81.
15. Dromey RG. "A Model for Software Product Quality." *IEEE Transactions on Software Engineering* 1995; 21:146–162.
16. Pfleeger SL, Atlee JM. *Software Engineering: Theory and Practice*, 4th ed. Upper Saddle River: Prentice Hall, 2009.
17. INFOCERT. http://www.infocert.org.
18. SASO. http://www.saso.org.pl.
19. Quality Assurance Institute. http://www.qaiglobalinstitute.com.
20. ISO/IEC 25000 System and Software Engineering—SQuaRE—Software Product Quality Requirements and Evaluation. Geneva, Switzerland: International Organization for Standardization, 2005–2013.
21. ISO/IEC 25051 Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Requirements for Quality of Commercial Off-the-Self (COTS) Software Products and Instructions for Testing. Geneva, Switzerland: International Organization for Standardization, 2006.
22. ISO/IEC 25023 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Measurement of System and Software Product Quality. Geneva, Switzerland: International Organization for Standardization; document in development.
23. Oracle E-Business Suite. http://www.oracle.com/us/solutions/oos/e-business-suite/overview/index.html.

24. SAP Business Suite. http://www54.sap.com/solution/lob/finance/software/business-suite-apps-hana/index.html.
25. ISO/IEC 25010 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. Geneva, Switzerland: International Organization for Standardization, 2011.
26. Selby R, editor. *Software Engineering: Barry Boehm's Lifetime Contributions to Software Development, Management and Research*. New York: Wiley/IEEE Press, 2007.
27. Boehm BW, Brown JR, Kaspar JR, Lipow ML, MacCleod G. *Characteristics of Software Quality*. New York: American Elsevier, 1978.
28. Eppler MJ, Helfert M. "A Framework for the Classification of Data Quality Costs and an Analysis of their Progression." In *International Conference on Information Quality; November 5–7, 2004*. Cambridge: Massachusetts Institute of Technology, 2004.
29. Fairley RE. *Managing and Leading Software Projects*. Hoboken, N.J.: Wiley IEEE Computer Society, 2009.
30. IEEE Standard 1012–2004. *IEEE Standard for Software Verification and Validation*. New York: IEEE Computer Society, 2004.
31. Krebs B. "Cyber Incident Blamed for Nuclear Power Plant Shutdown." *The Washington Post*, June 5, 2008.
32. Suryn W, Trudeau PO, Mazzetti C. "Information Systems and their Relationship to Quality Engineering." 17th Software Quality Management International Conference, April 6–8, 2009, Southampton, UK.
33. "Financial transaction," http://en.wikipedia.org/wiki/Financial_transaction.
34. Boutaba R, Xiao J. "Network Management: State of the Art." *The International Federation for Information Processing* 2002; 92:127–145.
35. Robertson J. "Step Two Designs: Definition of Information Management Terms." Available at http://www.steptwo.com.au/papers/cmb_definition/index.html. Accessed May 14, 2013.
36. ISO/IEC 15504 (SPICE) Information Technology—Process Assessment Series of Standards. Geneva, Switzerland: International Organization for Standardization, documents in development.
37. ISO/IEC 9126-1 Software Engineering—Product Quality—Part 1: Quality Model. Geneva, Switzerland: International Organization for Standardization, 2001.
38. ISO 9126-2 Software Engineering—Product Quality—Part 2: External Metrics. Geneva, Switzerland: International Organization for Standardization, 2003.
39. ISO 9126-3 Software Engineering—Product Quality—Part 3: Internal Metrics. Geneva, Switzerland: International Organization for Standardization, 2003.
40. ISO 9126-4 Software Engineering—Product Quality—Part 4: Quality in Use Metrics. Geneva, Switzerland: International Organization for Standardization, 2004.
41. Suryn W, Stambollian A, Dormeux JC, Bégnoche L. "Software Quality Engineering—Where to Find It in Software Engineering Body of Knowledge (SWEBOK)." 14th Software Quality Management International Conference; April 10–12, 2006, Southampton, UK.
42. ISO/IEC 12207 Information Technology—Software Life Cycle Processes. Geneva, Switzerland: International Organization for Standardization, 1995.
43. Bourque P, Dupuis R, Abran A, Moore JW, Tripp L, Wolff S. "Fundamental Principles of Software Engineering: A Journey." *Journal of Systems and Software* 2002; 62: 59–70.