

People make quality happen (or don't)

Jochen Ludewig, University of Stuttgart

Summary:

In a free market, every demand will result in a supply. So, if everybody wants high quality software, why don't we have it? This article analyses the actual demand, and the expectations of producers and customers in the software market. Various aspects of software quality are discussed, and a taxonomy is provided (chapter 2).

The actual need for high quality software, and the obstacles to producing it, are discussed in chapters 3 and 4. Chapter 5 addresses the relationship between price and quality. Finally, in chapter 6, the cultural aspects of high quality are presented.

Prof. Jochen Ludewig, Computer Science Department, Univ. of Stuttgart, Breitwiesenstr. 20, D-70565 Stuttgart, phone/fax +49-711-7816-354/380, ludewig@informatik.uni-stuttgart.de

1. Myths and reality: Do we really want high software quality?

myth: (...) a popular belief or tradition that has grown up around something or someone, especially one embodying the ideals and institutions of a society.

Merriam Webster's Collegiate Dictionary

1.1 Definitions

quality

- (1) The degree to which a system, component, or process meets specified requirements.
- (2) The degree to which a system, component, or process meets customer or user needs or expectations.

IEEE Std. 610-1990

"Quality" is an abstract term; it does not physically exist like an apple, a person or a building, but only as an attribute of some other object. In our context, it is the quality of software that we are talking about. Therefore, the topics of this article are **software quality** (or just **quality**) and high quality software (**HQS**).

The IEEE-definition above refers to all requirements, or expectations. That would include properties like purpose, price, availability, and time to market. Though these properties are very important, we do not regard them as aspects of quality in this paper. Hence a specification like "a text processing software which is immediately available on a Macintosh PowerBook under MacOS for less than 600 SFr" does not mention any quality. The reason to exclude such "qualities" is that they are usually not within the scope of quality assurance.

For simplicity's sake, all people whose business is producing software are called **programmers**. Therefore, only a small fraction of a programmer's activity is actually programming.

1.2 The myths about quality

At first glance, quality seems to be extremely popular. Most people would agree in sentences like

- a) Of course, we all want HQS (i.e. we all want to produce, buy, own, and use high quality software).
- b) Sometimes, quality competes with other goals, like costs and deadlines.
- c) Building software is very difficult, that is why we often fail to produce HQS.

These are myths. The truth is very different:

- a) Very few people really want HQS. They appreciate high quality in the same way they appreciate a warm day in May: It is something sent from heaven, and free. But as soon as they are asked to pay for it, high quality is hardly on the shopping list. More specifically, people do not want to produce HQS because it is too hard; and they do not want to buy it because it may be available too late, cost too much, or run too slow. And though they often would like to *have* it, they are – for the same reasons – not prepared to *maintain* it.
- b) Quality *always* competes with other goals, and it is usually given as much attention as the more important goals allow for. The more important goals are costs, and time to market. Those aspects of quality which are obvious to the customer, like functionality, performance, and ease of use, will not be completely neglected. The less visible qualities, like readability of the code, are the poor relatives who die starving in case food is tight.
- c) The statement about the difficulty of producing HQS is not wrong, but misleading. It is like stating that speaking Oxford English is more difficult than speaking Cockney. Those who speak Cockney will certainly agree, but others who never learned Cockney will not, because Oxford English is well defined in many books, it is used in numerous plays, on the radio, on TV, etc. The truth is that *changing* to another language is hard, and so is changing to a style of work which supports the creation of HQS. People who are used to it will experience a more pleasant work with fewer surprises, and more job satisfaction.

In general, we *do* know how to produce HQS, but we also know that doing it is hard, and usually not rewarding, and neither we nor our colleagues are used to do it. Therefore, we rather develop software using "the same procedure as last year".

Provided we had plenty of time and resources, we could apply all the techniques described in the textbooks, and after a while, we would certainly succeed. But we have to change our attitudes, and spend far more time on the project than we usually do. Therefore, the real problems are effort, costs, and indolence.

1.3 What does it take to make a programmer build high quality software?

Many companies have tried to make their programmers produce HQS, but with little success. Two reliably unsuccessful approaches are:

- Just tell the programmers that you want high quality.
- Engrave your appreciation of quality in stone or stainless steel, and put it into the main lobby.

It takes more to change the habits. A person will do a certain job only if

- the job is within his or her ability (set A in fig. 1) and
- there is some benefit from doing the job, either by a positive response from the environment which recognises the results ("rewarding work", set E), or by the fun from doing it ("pleasant activity", set F).

The diagram indicates that the set of all tasks we like to do (L) is mainly determined by our abilities; some of the tasks in L may even be fun to do. This can be expressed by

$$F \cap L \cap A$$

Provided we have sufficient time, we will do all the tasks which are in F, or both in L and E. Hence, the tasks we do are the set (shaded in fig. 1)

$$D = F \cup (L \cap E)$$

Most of the people who are too busy suffer from a very large set D; those who are frustrated in their job suffer from the fact that both F and (L ∩ E) are small, or even empty.

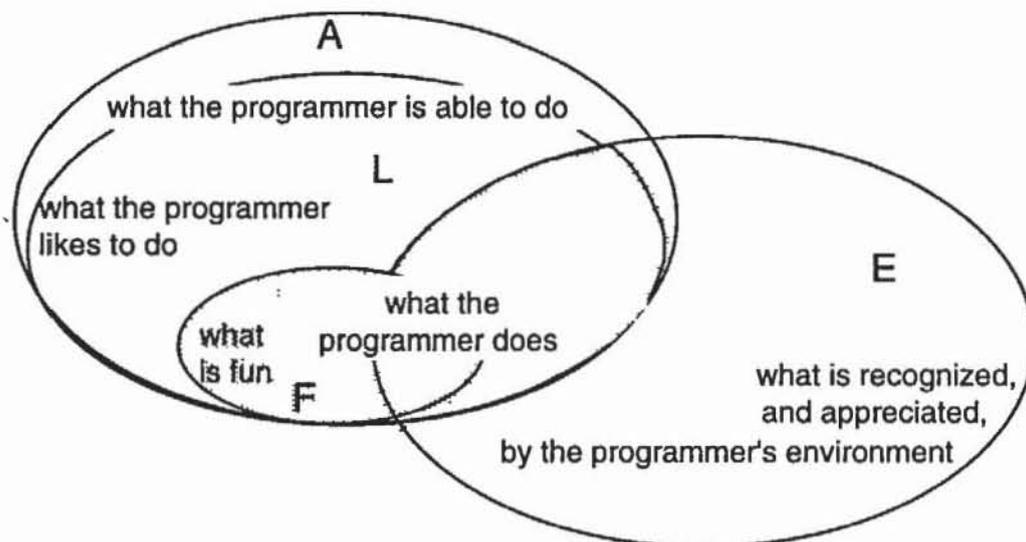


Figure 1: Tasks the programmers do

Many deficiencies in Software Engineering can be explained by this simple diagram: Skill *and* motivation are prerequisites for every job. In many problem areas, both are missing.

If you want people to produce HQS, make sure that

- they really know how to do it, not only from books, but also from practical experience, and
- any effort towards HQS will be supported and appreciated by their environment, whereas all obstacles will be removed.

If you can make it fun to produce HQS, you have won.

2. What is high quality made of?

2.1 A taxonomy of software quality

Quality can be decomposed into many special qualities. Fig. 2 shows such a taxonomy, i.e. a tree-structured decomposition. The advantage of any such refinement is that it can be used both as a checklist for specifying high quality, and for evaluating software and software projects.

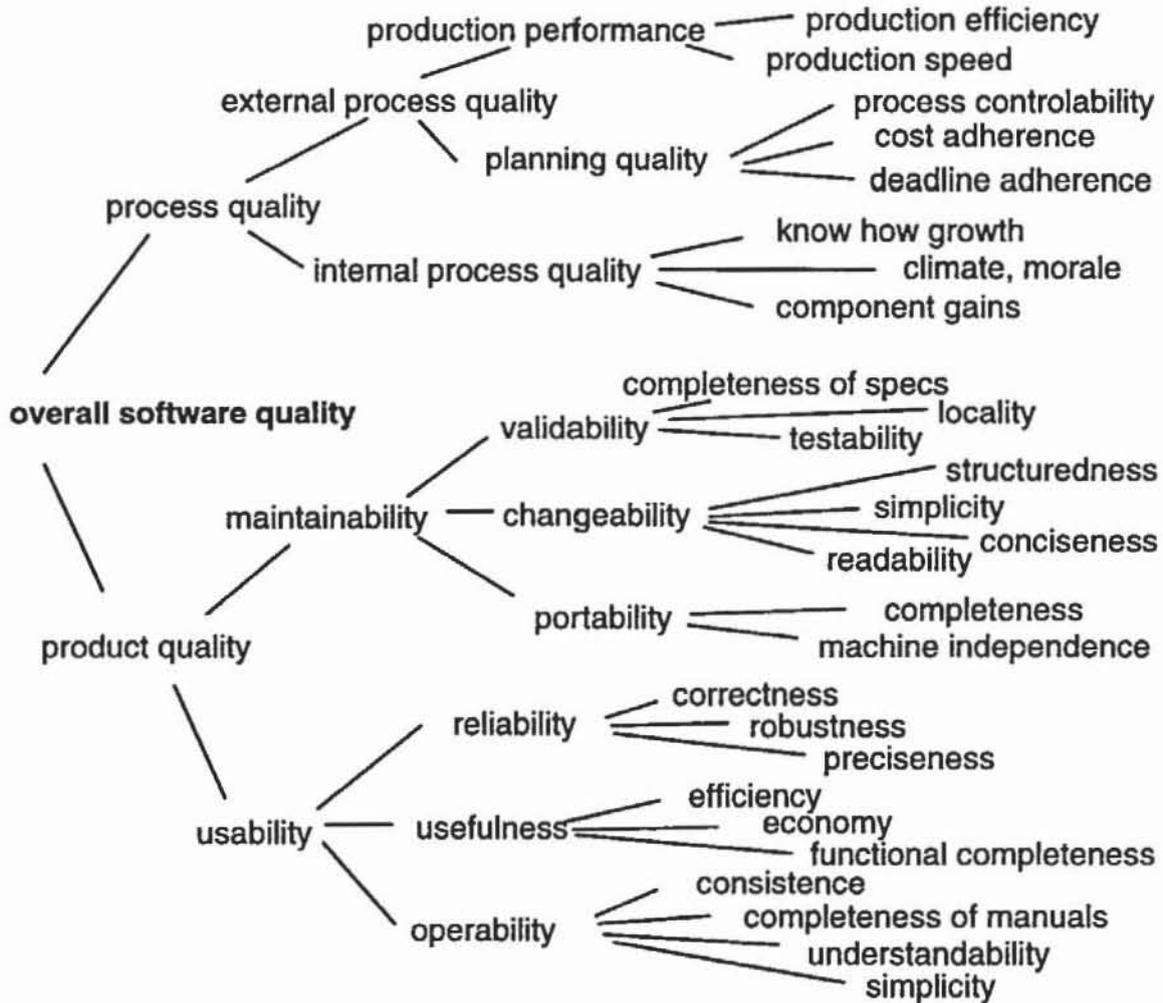


Figure 2: A software quality taxonomy

2.2 How to measure software quality

Measuring software quality is not a simple task. Many attempts to define mathematical mappings ("metrics") which can be cast into automatic tools have failed. For the time being, we can either measure properties whose meaning is questionable (like McCabe's Cyclomatic Complexity), or we can try to figure out important properties (like simplicity), which cannot be measured by means of automatic tools.

Sad as the current situation is, it still offers a very simple advice: There is no hope at all for the fully automatic general analyser which is based on standardised metrics. Choose the properties you wish to know about, define scales to describe them, and define procedures to

collect the data. Such procedures will in almost any case involve experts who have to do the judgement.

3. The need for high quality software

The discussion above indicates that we hardly want to produce, or afford, HQS. Is there any reason at all to insist on HQS?

This question is rarely asked, or immediately answered by religious commandments ("Thou shall produce highest quality"). Such arguments do not count. Software quality is not an end in itself, but a step on the way to efficient Software Engineering.

software engineering	
(1)	The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
(2)	The study of approaches as in (1).

IEEE Std. 610-1990

The ultimate goal of Software Engineering (like any engineering) is to achieve highest benefits at lowest costs. Software quality is desirable exactly to the degree it contributes to that goal. "Costs", however, include everything, not only coding, or software construction, but also user satisfaction, maintenance, risks, the additional costs of components which have to be developed twice because they are not reusable, and so on. We do not just talk about the mere costs of production, but about total expenses. Fig. 3 shows which costs contribute to the total costs of software. The very last line, i.e. the risks, may well exceed everything else.

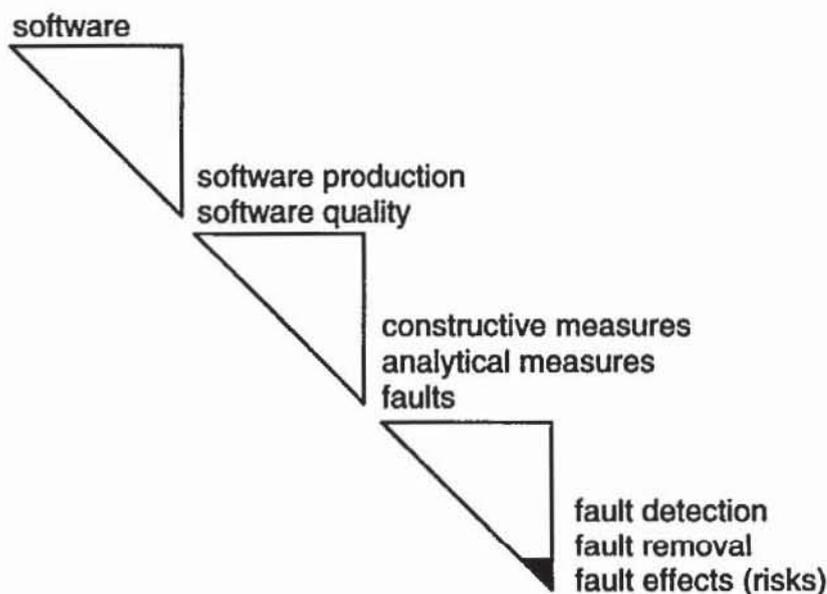


Figure 3: The high cost of saving on quality

In fig. 3, "software production" means only constructive activities, like coding or integration, without constructive and analytical QA, like training and test. The distinction between production and constructive QA is obviously difficult.

As far as benefits are concerned, HQS will be superior due to

- increased user satisfaction
- better success in the software market
- increased life span of the software
- higher appreciation of the producer

Costs are reduced due to

- less maintenance
- reduced risk from failure, and from corrective actions
- software reuse in following projects

Note that the benefits of HQS grow with the software life span. Compared to average quality software, HQS has an increased life span, because people like to use it, and do not look for other systems too soon. This means that those who produce HQS gain twice: First by better success, and second by cost reduction due to the postponed need for a follow-up-system.

Other benefits are hard to estimate. When HQS has been used for a while, people will reduce their efforts on unreliability-driven activities, like implementing the same algorithm again and again, etc.

None of these effects works instantaneously; producing HQS is an investment. The profit will not be reaped before some delay. Organisations which are unable to invest in software quality, and then to wait for some time, say two years, are like stupid (or extremely poor) people who eat up the seed corn instead of harvesting the yield.

These points can be summarised as follows:

- The ultimate goal of Software Engineering is to maximise the net gain, i.e. the difference between profits and costs.
- Expenditures for HQS are justified by their effects on the proceeds. Quality should be increased as long as the effect on the proceeds is positive, but no longer.

These points are illustrated in figure 4. The basic relationships are:

$$G = f(C_Q) = P(C_Q) - (C_Q + C_M(C_Q))$$

$$\forall C_Q \geq 0 : C(\Omega) = C(C_Q), \text{ i.e. } \Omega \text{ is the optimum investment in software quality.}$$

In any specific project, Ω must be known in order to make $C_Q = \Omega$. In general, that is a difficult problem. In most projects, however, we know at least that $C_Q \ll \Omega$. Then, C_Q should be increased. Since $G(C_Q)$ has a flat top, it does not really matter whether we manage to give a precise estimate of Ω . Since we usually underestimate the life span of the software system, it is safe to use a higher value of C_Q .

4. How to prevent the production of high quality software

As mentioned in 1.2, producing HQS is not (primarily) a technical problem. Provided we have enough programmers, time, and money, there are numerous textbooks and consultants which can teach us how to do it. Still, we do not see HQS too often. Programmers and their managers find many ways to escape from HQS. Here is a "How to-list" for those who want to prevent the production of HQS (Table 1):

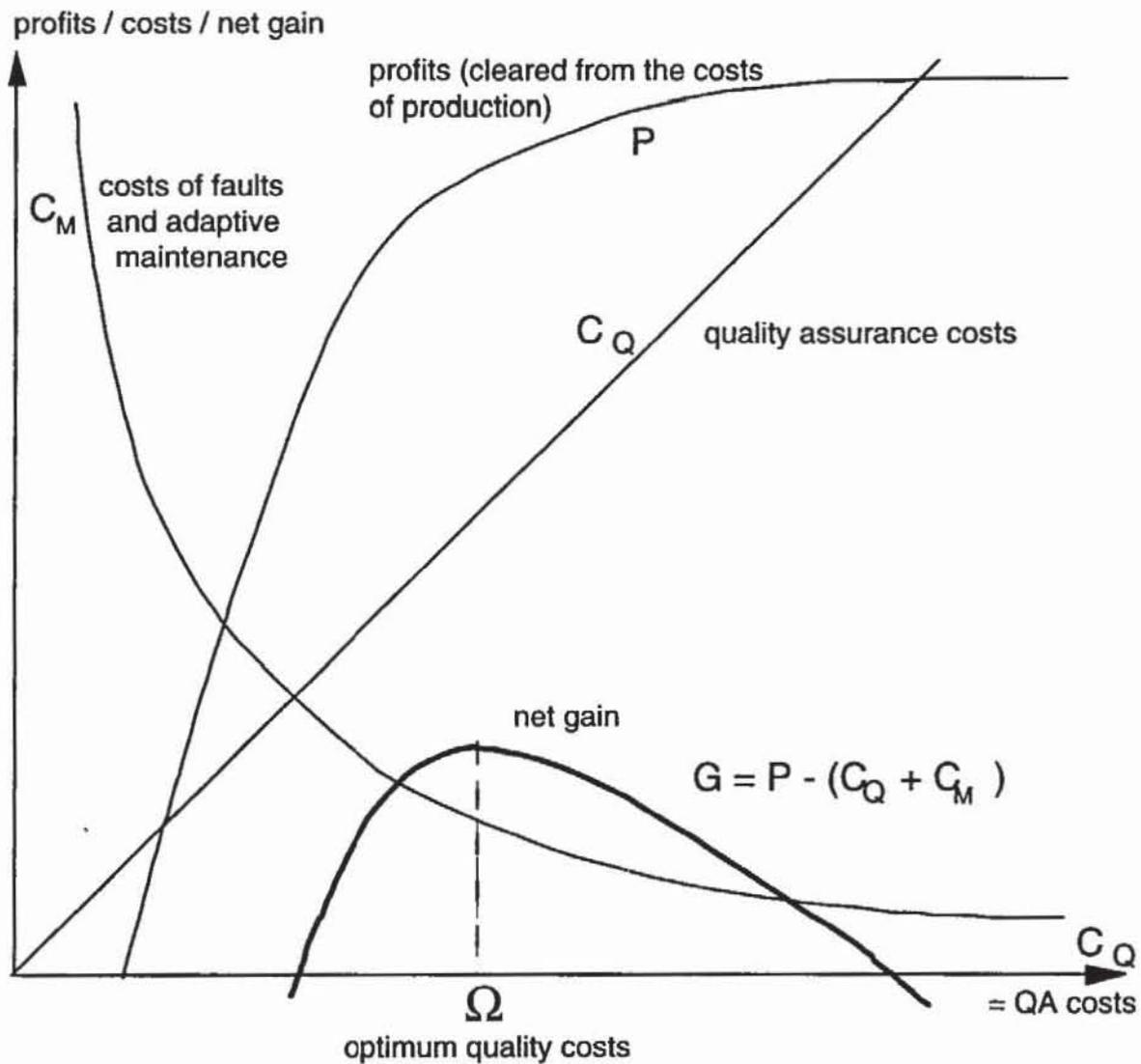


Figure 4: Profits, costs, and net gain of software (cp. fig. 3)

Approach	prevents HQS because	better:
a) Wishful thinking: Simply deny that you have a software problem.	As long as you do not open your eyes, you cannot understand your problems, and hence cannot solve them.	Start from a serious analysis of the projects, their strengths and weaknesses, in order to improve them.
b) Reduce your views and decisions to a binary choice.	In Software Engineering, many ideal solutions are not available, or not even possible. Then, a binary decision means no change at all.	When the ideal solution is not possible (like a strictly formal specification, e.g.), we may find a compromise which is still better than the traditional approach.

(Table 1, continued)

Approach	prevents HQS because	better:
c) Set goals which cannot be reached.	An unrealistic goal (like zero defect software, e.g.) does not imply any obligation for the programmers. Therefore, they won't care.	Set goals which are ambitious, but realistic; and plan how, and when, to reach them. Define milestones and criteria.
d) Believe in miracles.	Those who believe in miracles do not feel responsible for changing (a small piece of) the world, because the miracle will do it.	Expect to work with very similar methods, languages, tools, and people next year. Nothing will change if you don't change it.
e) Give up any hope to improve your programmers' performance.	Most programmers never learned their profession properly. When they are not regularly coached, they will get lost (mentally).	Programmers should get a feedback of their performance regularly, and should maintain their qualification. Then, they will be open for better tools and methods.
f) Avoid collecting any data.	As long as there is no valid data, quality is a matter of opinions, or taste.	Collect (simple) data systematically, and make them available for evaluations, in order to recognise and prove influences to quality.
g) Don't use software configuration management.	As long as it is impossible to identify, and reconstruct the software or its components, it is never clear to which object statements about quality etc. refer.	Well organised, painstaking configuration management will be the foundation of all improvements towards HQS.
h) Introduce tools, but without much evaluation and planning. Hand them over like gifts to make sure the programmers won't like them.	Tools are useful only if they fit to the methods which are applied. Introducing tools is a time consuming and expensive task.	Very few changes in Software Engineering start with new tools. Change the methods and procedures, and introduce tools only when there is a clear demand.
i) Make sure that the experience of one project is not used for any following project.	Improvements take longer than projects. As long as the world starts all over again with every new project, there is little chance of improvements.	Update your plans and schedules while the project is proceeding, and analyse the reasons of successes and failures. Make these experiences available to everybody else in your organisation.

5. The price of high quality software

High quality products cost much more than poor ones. While a tinker buys a plane for 10 SFr, a cabinet-maker will pay more than 100 SFr for a thing which looks very similar. What is the difference? The expensive one is of high quality, while the other one is not. Both customers try to get the best value. But the craftsman is sufficiently skilled and experienced to know that he will work much more efficiently with an excellent plane.

Producing high quality products requires *both* a competent producer and a competent customer who insists on high quality. Without the customer, the producer cannot survive.

Software systems are among the most complex systems which have been created by man. There are other complex systems, like organisations, cities, vessels, and aeroplanes. But all these systems are interactively controlled, i.e. deficiencies of the general design are compensated by human interaction. This is not possible for software: If an operating system kernel would fail in one out of one million operations, it would be useless, we could not call for human interaction to overcome the difficulty.

Software cannot become less complex, because it models the reality. A system used for computing the tax is necessarily at least as complex as the tax system is. The same argument applies to real time systems, application software, and most other software systems. Therefore, software will never be simple, and it will always remain hard to produce it. It will be even harder when the expectations of quality rise.

Such software cannot be cheap. HQS will inevitably be more expensive than most software systems are today. And that implies a very different understanding and behaviour of producers and customers in the software market.

Today, we buy software for little money. We are not surprised to find disclaimers saying that nothing at all is guaranteed, in particular no consistence with any other software. Maybe we use it for a while, then we are fed up, and after some trouble due to a new version of the operating system, we replace the package by another one.

This description holds for the low end software, i.e. for software which is used on PCs and – to a certain degree – on workstations. In the high end market, the relations between producers and customers are closer, which helps to overcome some of the difficulties. But the general problem is still the same.

What would a HQS market look like? The producers of HQS take full responsibility, otherwise it is not HQS. And the customers do not accept less after spending big money. They use the software for a long time. Since they use nothing but well engineered HQS, they can combine all their systems. When they change their hardware, the software is not affected, because it is based on stable, standardised interfaces.

Note that in such a situation, software is no longer *soft*. it is just as stable as any machine, building, or other object of investment. And just as reliable.

6. High quality software: effort versus culture

effort: (...) **1:** conscious exertion of power: hard work

culture: (...) **5 a:** the integrated pattern of human knowledge, belief, and behaviour that depends upon man's capacity for learning and transmitting knowledge to succeeding generations **b:** the customary beliefs, social forms, and material traits of a racial, religious, or social group **c:** the set of shared attitudes, values, goals, and practices that characterises a company or corporation

Merriam Webster's Collegiate Dictionary

In a mechanical workshop, people produce mechanical parts at a certain level of quality. Though they may sometimes be required to achieve extraordinary preciseness, high quality is usually not the effect of special efforts and inspections, but is just normal. It is the result of many favourable conditions, like:

- the craftsmen's skill, which is based on decent training
- the high quality of their tools
- generally accepted standards for components, materials, procedures, and notations
- the designers' knowledge, which is also based on a profound education
- an approved working organisation
- customers who ask for high quality, and are prepared to pay its price
- and, last but not least, the attitudes of all people in the workshop, and the mutual social control between them.

All these conditions together form a **culture**.

Culture is never a private matter. Imagine a group of people who should work together, but rarely do so because their working hours differ widely, and there is no obvious pattern in anybody's behaviour. If they really have to meet, they have to make arrangements, which often fail, because they are not used to be punctual. Then, no single person can improve the situation.

But if most of them agree to meet at a certain time, and actually do, they have raised their culture. After a while, they will find it most comfortable to be members of a reliable group, and wonder why other people do not do it in the same way: They do not need any special arrangements, because their co-operation goes *without saying*.

That is culture: The habits and attitudes which go without saying (and even without conscious thinking). Culture differs from effort because it is much easier to do things well when you are used to do so, and the environment does not expect anything else, and all the procedures, methods, and tools are based on the common understanding of high quality.

Effort is contributed by an individual, while culture is achieved by a group. Any extra effort will stop after some time, because it means overload; culture is stable, provided there is no major disturbance. See fig. 5 which shows the effects of effort on the total quality (which is a weighted mix of relevant qualities of a product or process). Increased effort results in increased quality, but only for a particular project. And there is a limit which will not be surpassed: The general conditions do not allow for better quality. If, on the other hand, the cultural level is raised (along the arrows), the effect is lasting, and does not require a particular effort. Then, high quality is achieved easily.

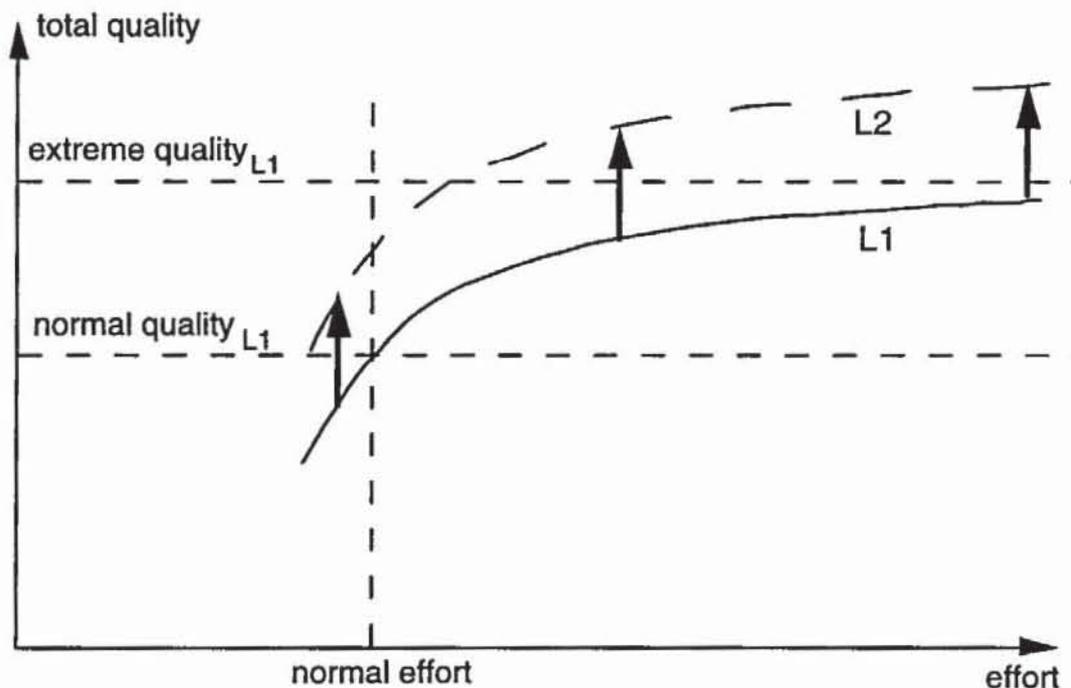


Figure 5: Raising the effort versus improving the culture

7. Conclusion

HQS is not primarily a technical problem, but a matter of priorities. We can achieve high quality in a particular situation by an extraordinary effort, but in order to obtain a stable, lasting improvement, we have to raise the cultural level. Such a step implies training, project planning, quality assurance, modern methods and tools, and an affirmative feedback to those who switched to the desired habits.

High quality will never be free; we have to pay for it. But working towards high quality will become easy and pleasant when everybody is used to do so.

Acknowledgements

Marcus Deininger has checked an earlier version of this paper, and contributed a couple of improvements.

This article is partially based on the author's earlier work, in particular on an article of 1987¹, on a book first published in 1988², and also on the author's lectures on Software Engineering (unpublished). No other material was used.

¹J. Ludewig: Software Engineering (Software und Qualitätssicherung – Versuch einer Annäherung. in K. Tomica (ed.): Software-Qualitätssicherung 1987. SAQ, Bern, pp. 7-25

²K. Frühauf, J. Ludewig, H. Sandmayr: Software-Projekt-Management und -Qualitätssicherung. vdf, Zürich, and Teubner, Stuttgart, 2nd ed., 1991